The Correct Way to Overload Functions in Python

MARTIN May 31, 2021 | #Python

Function overloading is a common programming pattern which seems to be reserved to statically-typed, compiled languages. Yet there's an easy way to implement it in Python with help of *Multiple Dispatch* or as it's called in Python *multi-methods*.

Overloading

First things first - you might be asking, how can we implement method overloading in Python when we all know that it's not possible? Well, even though Python is dynamically-typed language and therefore cannot have proper method overloading as that requires the language to be able to discriminate between types at compile-time, we can still implement it in a bit different way that is suitable for dynamically-typed languages.

This approach is called *Multiple Dispatch* or *multimethods*, where the interpreter differentiates between multiple implementations of a function/method at runtime based on dynamically determined types. To be more precise, the language uses types of arguments passed to a function during its invocation to dynamically choose which one of the multiple function implementations to use (or dispatch).

Now you might be thinking: *"Do we really need this though? If it can't be implemented normally, maybe we shouldn't use it in Python..."* Yea, valid point, but there are good reasons to want to implement some form of function/method overloading in Python. It's powerful tool that can make code more concise, readable and minimise its complexity. Without multimethods though, the *"obvious way"* to do this is using type inspection with `isinstance()`. This is very ugly, brittle solution that is closed to extension and I would call it an anti-pattern.

Besides that, there already is method overloading in Python for operators and methods like `len()` or `new()` using so-called *dunder* or *magic* methods (see docs here) and we all use that quite often, so why not use proper overloading for *all* the function, right?

So, now we know that we can kind-of implement overloading in Python, so how exactly do we do that?

Single Dispatch

Above we spoke about *Multiple Dispatch*, but Python doesn't support this out-of-the box, or in other words *Multiple Dispatch* is not a feature of Python standard library. What is available to us however, is called *Single Dispatch*, so let's begin with this simpler case first.

The only actual difference between *multi* and *single* dispatch is number of arguments which we can overload. So, for this implementation in standard library it's just one.

The function (and decorator) that provides this feature is called `singledispatch` and can be found in `functools` module.

This whole concept is best explained with some examples. There are many *"academic"* examples of overloading functions (geometric shapes, addition, subtraction...) that we've probably all seen already. Rather than going over that, let's see some practical examples. So, here's first example for `singledispatch` to format dates, times and datetimes:

```
from functools import singledispatch
from datetime import date, datetime, time

@singledispatch
def format(arg):
    return arg

@format.register
def _(arg: date):
    return f"{arg.day}-{arg.month}-{arg.year}"
```

```
@format.register
def _(arg: datetime):
    return f"{arg.day}-{arg.month}-{arg.year} {arg.hour}:{arg.minute}:{arg.second}"

@format.register(time)
def _(arg):
    return f"{arg.hour}:{arg.minute}:{arg.second}"

print(format("today"))
# today
print(format(date(2021, 5, 26)))
# 26-5-2021
print(format(datetime(2021, 5, 26, 17, 25, 10)))
# 26-5-2021 17:25:10
print(format(time(19, 22, 15)))
# 19:22:15
```

We begin by defining the base `format` function that is going to be overloaded. This function is decorated with `@singledispatch` and provides base implementation, which is used if no better options is available. Next, we define individual functions for each type that we want to overload - in this case `date`, `datetime` and `time` - each of these have name `_` (underscore) because they will be called (dispatched) through the `format` method anyway, so no need to give them useful names. Each of them is also decorated with `@format.register` which attaches them to the previously mentioned `format` function. Then, to make it possible to differentiate between types, we have two options - we can use type annotations - as demonstrated in first two cases or explicitly add the type to decorator as with the last one from the example.

In some cases it might make sense to use same implementation for multiple types - for example for number types such as `int` and `float` - for these situations decorator stacking is allowed, meaning that you can list (stack) multiple `@format.register(type)` lines to associate a function with all the valid types.

Besides ability to overload basic functions, `functools` module contains also `singledispatchmethod` that can be applied to methods of a class. Example of that could be the following:

```
from functools import singledispatchmethod
from datetime import date, time

class Formatter:
    @singledispatchmethod
    def format(self, arg):
        raise NotImplementedError(f"Cannot format value of type {type(arg)}")

    @format.register
    def _(self, arg: date):
        return f"{arg.day}-{arg.month}-{arg.year}"

    @format.register
    def _(self, arg: time):
        return f"{arg.hour}:{arg.minute}:{arg.second}"

f = Formatter()
print(f.format(date(2021, 5, 26)))
# 26-5-2021
print(f.format(time(19, 22, 15)))
# 19:22:15
```

Multiple Dispatch

Oftentimes *Single Dispatch* won't be sufficient and you might need the proper *Multiple Dispatch* functionality. This is available from `multipledispatch` module which can be found here and can be installed with `pip install multipledispatch`.

This module and it's decorator - `@dispatch`, behaves very similarly to the `@singledispatch` in the standard library. Only actual difference is that it can take multiple types as arguments:

```
from multipledispatch import dispatch

@dispatch(list, str)
```

```
def concatenate(a, b):
    a.append(b)
    return a

@dispatch(str, str)
def concatenate(a, b):
    return a + b

@dispatch(str, int)
def concatenate(a, b):
    return a + str(b)


print(concatenate(["a", "b"], "c"))
# ['a', 'b', 'c']
print(concatenate("Hello", "World"))
# HelloWorld
print(concatenate("a", 1))
# a1
```

The above snippet shows how we can use `@dispatch` decorator to overload multiple arguments, for example to implement concatenation of various types. As you probably noticed, with `multipledispatch` library we didn't need to define and register base function, rather we created multiple functions with same name. If we wanted to provide base implementation, we could use `@dispatch(object, object)` which would catch any non-specific argument types.

The previous examples shows proof-of-concept, but if we wanted to really implement such `concatenate` function, we would need to make it much more generic. This can be solved with use of union types. In this specific example we could change the first function as follows:

```
@dispatch((list, tuple), (str, int))
def concatenate(a, b):
    return list(a) + [b]

print(concatenate(["a", "b"], "c"))
```

```
# ['a', 'b', 'c']
print(concatenate(("a", "b"), 1))
# ['a', 'b', 1]
```

This would make it so that first argument of the function could be any of `list` or `tuple`, while second one would be `str` or `int`. This is already much better than the previous solution, but it can be further improved using abstract types. Instead of listing all the possible sequences, we can use `Sequence` abstract type (assuming that our implementation can handle it) which covers things like `list`, `tuple` or `range`:

```
from collections.abc import Sequence

@dispatch(Sequence, (str, int))
def concatenate(a, b):
    return list(a) + [b]
```

If you want to take this approach, then it's good to take a look at `collections.abc` module and see which container data-type best suits your needs. Mostly to make sure that your function will be able to handle all the types that fall into the chosen container.

All this mixing and matching of argument types is convenient, but can also cause ambiguities when choosing suitable function for some specific set of parameters. Fortunately, `multipledispatch` provides `AmbiguityWarning` which is raised if ambiguous behaviour is possible:

```
test_multipledispatch:10: AmbiguityWarning:
Ambiguities exist in dispatched function some_func

The following signatures may result in ambiguous behavior:
    [str, object], [object, str]


Consider making the following additions:

@dispatch(str, str)
```

```
def some_func(...)
```

Closing Thoughts

In this article we went over a simple, yet powerful concept which I rarely see being used in Python, which is a shame considering that it can greatly improve code readability and get rid of anti-patters like type inspection using `isinstance()`. Also, I hope you would agree that this approach to function over-loading should be considered the *"obvious way"* and I hope that you will make use of it when needed.

If you want to dive deeper into this topic and get your hands dirty you can implement multimethods yourselves as shown in Guido's article - this can be a good exercise to understand how multiple dispatch actually works.

Finally, I should also probably mention that this article omits examples of the well-known operator overloading which I mentioned in the beginning as well as some approaches for overloading constructors for example using factories. So, in case that's what you're looking for, go check out these links/resources, which give good overview on there topics.
