Function Overloading with functools.singledispatch

Python fairly recently added partial support for function overloading in Python 3.4. They did this by adding a neat little decorator to the functools module called singledispatch. This decorator will transform your regular function into a single dispatch generic function. Note however that singledispatch only happens based on the first argument's type. Let's take a look at an example to see how this works!

```
from functools import singledispatch


@singledispatch
def add(a, b):
    raise NotImplementedError('Unsupported type')


@add.register(int)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)


@add.register(str)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)


@add.register(list)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)


if __name__ == '__main__':
    add(1, 2)
    add('Python', 'Programming')
    add([1, 2, 3], [5, 6, 7])
```

Here we import singledispatch from functools and apply it to a simple

function that we call add. This function is our catch-all function and will only

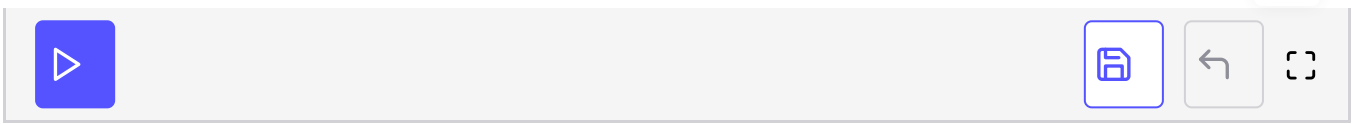
get called if none of the other decorated functions handle the type passed. You will note that we currently handle integers, strings and lists as the first argument. If we were to call our add function with something else, such as a dictionary, then it would raise a NotImplementedError.

Try running the code yourself. You should see output that looks like this:

```
First argument is of type  <class 'int'>
3
First argument is of type  <class 'str'>
PythonProgramming
First argument is of type  <class 'list'>
[1, 2, 3, 5, 6, 7]
Traceback (most recent call last):
  File "overloads.py", line 30, in <module>
    add({}, 1)
  File "/usr/local/lib/python3.5/functools.py", line 743, in wrapper
    return dispatch(args[0].__class__)(*args, **kw)
  File "overloads.py", line 5, in add
    raise NotImplementedError('Unsupported type')
NotImplementedError: Unsupported type
```

As you can see, the code works exactly as advertised. It calls the appropriate function based on the first argument's type. If the type isn't handled, then we raise a NotImplementedError. If you want to know what types we are currently handling, you can add the following piece of code to the end of the file, preferable before the line that raises an error:

```
print(add.registry.keys())
```



This will print out something like this:

```
dict_keys([<class 'str'>, <class 'int'>, <class 'list'>, <class 'object'>])
```

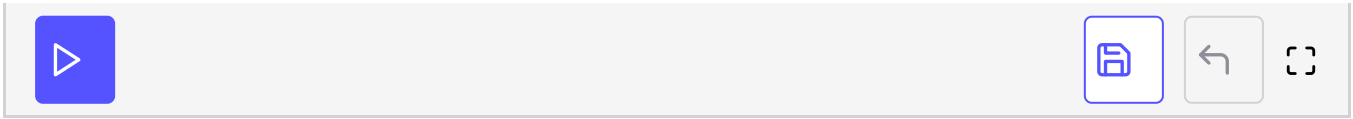
This tells us that we can handle strings, integers, lists and objects (the default). The singledispatch decorator also supports decorator stacking. This allows us to create an overloaded function that can handle multiple types. Let's take a look:

```
from functools import singledispatch
from decimal import Decimal


@singledispatch
def add(a, b):
    raise NotImplementedError('Unsupported type')


@add.register(float)
@add.register(Decimal)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)


if __name__ == '__main__':
    add(1.23, 5.5)
    add(Decimal(100.5), Decimal(10.789))
```



This basically tells Python that one of the add function overloads can handle float and decimal.Decimal types as the first argument. If you run this code, you should see something like the following:

```
First argument is of type  <class 'float'>
6.73
First argument is of type  <class 'decimal.Decimal'>
111.28899999999999997015720510
dict_keys([<class 'float'>, <class 'int'>, <class 'object'>, <class 'decimal.Decimal'>
```

You may have noticed this already, but because of the way all these functions were written, you could stack the decorators to handle all the cases in the previous example and this example into one overloaded function. However, in a normal overloaded case, each overload would call different code instead of doing the same thing.