Single_dispatch

May 7, 2021

0.1 Singledispatch

When dispatching based on the type is required, the most simple approach possible would something similar to the following:

Note: DO NOT USE THIS APPROACH IN REAL CODE. THIS IS TO SHOW WHAT SHOULD ***NOT*** *BE USED*

[3]:
```
class Base:
    def __init__(self, value):
        self._value = value

class A(Base):
    def __init__(self, value, a):
        super().__init__(value)
        self._a = a

class B(Base):
    def __init__(self, value, b):
        super().__init__(value)
        self._b = b

class C(Base):
    def __init__(self, value, c):
        super().__init__(value)
        self._c = c

def print_a(a):
    print(f'A class with {a._value} and {a._a}')

def print_b(b):
    print(f'B class with {b._value} and {b._b}')

def print_c(c):
    print(f'C class with {c._value} and {c._c}')

def my_print(something):
    if isinstance(something, A):
        print_a(something)
```

```
    elif isinstance(something, B):
        print_b(something)
    elif isinstance(something, C):
        print_c(something)
    else:
        raise ValueError(f'Invalid argument type {something}')
```

[4]:
```
a = A(1, 'A')
b = B(2, ['wololo'])
c = C(3, {'one': 'uno'})

my_print(a)
my_print(b)
my_print(c)
my_print(42)
```

```
A class with 1 and A
B class with 2 and ['wololo']
C class with 3 and {'one': 'uno'}
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-765d1a290d20> in <module>
      6 my_print(b)
      7 my_print(c)
----> 8 my_print(42)

<ipython-input-3-5ba6b2dabb4d> in my_print(something)
     35         print_c(something)
     36     else:
---> 37         raise ValueError(f'Invalid argument type {something}')

ValueError: Invalid argument type 42
```

The code in `my_print` is really bad. You might be thinking that could be optimized by using a dictionary, like in the following example

[5]:
```
def my_print_bis(something):
    calls = {
        A: lambda x: print_a(x),
        B: lambda x: print_b(x),
        C: lambda x: print_c(x),
    }

    try:
        calls[type(something)](something)
    except IndexError:
```

```
        raise ValueError(f'Invalid argument type {something}')
```

[6]:
```
my_print_bis(a)
my_print_bis(b)
my_print_bis(c)
my_print_bis(42)
```

```
A class with 1 and A
B class with 2 and ['wololo']
C class with 3 and {'one': 'uno'}
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-6-f2c6df5d53e6> in <module>
      2 my_print_bis(b)
      3 my_print_bis(c)
----> 4 my_print_bis(42)

<ipython-input-5-20522eb516fe> in my_print_bis(something)
      7
      8     try:
----> 9         calls[type(something)](something)
     10     except IndexError:
     11         raise ValueError(f'Invalid argument type {something}')

KeyError: <class 'int'>
```

However, these are all still wrong because now we are checking exact types, and not taking into account inheritance.

Now, to use properly the `singledispatch` decorator we need to do the following:

[7]:
```
from functools import singledispatch

@singledispatch
def my_print_bis_bis(value):
    raise ValueError(f'Invalid argument type {something}')

@my_print_bis_bis.register(A)
def _(a):
    print_a(a)

@my_print_bis_bis.register(B)
def _(b):
    print_b(b)

@my_print_bis_bis.register(C)
```

```
def _(c):
    print_c(c)
```

[8]:
```
my_print_bis_bis(a)
my_print_bis_bis(b)
my_print_bis_bis(c)
my_print_bis_bis(42)
```

```
A class with 1 and A
B class with 2 and ['wololo']
C class with 3 and {'one': 'uno'}
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-1e705eca8235> in <module>
      2 my_print_bis_bis(b)
      3 my_print_bis_bis(c)
----> 4 my_print_bis_bis(42)

/opt/anaconda3/lib/python3.7/functools.py in wrapper(*args, **kw)
    838                             '1 positional argument')
    839
--> 840         return dispatch(args[0].__class__)(*args, **kw)
    841
    842     funcname = getattr(func, '__name__', 'singledispatch function')

<ipython-input-7-2166b481a3fe> in my_print_bis_bis(value)
      3 @singledispatch
      4 def my_print_bis_bis(value):
----> 5     raise ValueError(f'Invalid argument type {something}')
      6
      7 @my_print_bis_bis.register(A)

NameError: name 'something' is not defined
```

Do not use the singledispatch with methods, since it uses the first arguments type to know which registered function needs to be called, and it will always be self To use it in that way an external method that swaps the first and second parameter needs to be used.

[9]:
```
class D(Base):
    def __init__(self, value, d):
        super().__init__(value)
        self._d = d

    def do_something(self, letter):
        something_to_do(letter, self)
```

```
def print_d(d):
    print(f'D class with {d._value} and {d._d}')

@singledispatch
def something_to_do(letter, clss):
    raise TypeError(f'No idea what you want me to do')

@something_to_do.register(A)
def _(letter, clss):
    return print_a(letter)

@something_to_do.register(B)
def _(letter, clss):
    return print_b(letter)

@something_to_do.register(C)
def _(letter, clss):
    return print_c(letter)

@something_to_do.register(D)
def _(letter, clss):
    return print_d(letter)
```

[10]:
```
d = D(4, (1,2,3,4))

d.do_something(a)
d.do_something(b)
d.do_something(c)
d.do_something(d)
```

```
A class with 1 and A
B class with 2 and ['wololo']
C class with 3 and {'one': 'uno'}
D class with 4 and (1, 2, 3, 4)
```

[ ]: