Arpit Bhayani FOLLOW
Backend @Unacademy • Data @Amazon • Platform @Practo | Writes about Language internal...

Overload Functions in Python

Published Feb 07, 2020

Function overloading is the ability to have multiple functions with the same name but with different signatures/implementations. When an overloaded function `fn` is called, the runtime first evaluates the arguments/parameters passed to the function call and judging by this invokes the corresponding implementation.

```
int area(int length, int breadth) {
  return length * breadth;
}

float area(int radius) {
  return 3.14 * radius * radius;
}
```

In the above example (written in C++), the function `area` is overloaded with two implementations; one accepts two arguments (both integers) representing the length and the breadth of a rectangle and returns the area; while the other function accepts an integer radius of a circle. When we call the function `area` like `area(7)` it invokes the second function while `area(3, 4)` invokes the first.

Why no Function Overloading in Python?

Python does not support function overloading. When we define multiple functions with the same name, th , there will namespaces by invoking functions `locals()` and `globals()` , which returns local and global namespace respectively.

```
def area(radius):
  return 3.14 * radius ** 2

>>> locals()
{
  ...
  'area': <function area at 0x10476a440>,
  ...
}
```

Calling the function `locals()` after defining a function we see that it returns a dictionary of all variables defined in the local namespace. The key of the dictionary is the name of the variable and value is the reference/value of that variable. When the runtime encounters another function with the same name it updates the entry in the local namespace and thus removes the possibility of two functions co-existing. Hence python does not support Function overloading. It was the design decision made while creating language but this does not stop us from implementing it, so let's overload some functions.

Implementing Function Overloading in Python

We know how Python manages namespaces and if we would want to implement function overloading, we would need to

- manage the function definitions in a maintained virtual namespace
- find a way to invoke the appropriate function as per the arguments passed to it

To keep things simple, we will implement function overloading where the functions with the same name are distinguished by the **number of arguments** it accepts.

Wrapping the function

We create a class called `Function` that wraps any function and makes it callable through an overridden `__call__` method and also exposes a method called `key` that returns a tuple which makes this function unique in entire codebase.

```python
from inspect import getfullargspec

class Function(object):
  """Function is a wrap over standard python function.
  """
  def __init__(self, fn):
    self.fn = fn

  def __call__(self, *args, **kwargs):
    """when invoked like a function it internally invokes
    the wrapped function and returns the returned value.
    """
    return self.fn(*args, **kwargs)

  def key(self, args=None):
    """Returns the key that will uniquely identify
    a function (even when it is overloaded).
    """
    # if args not specified, extract the arguments from the
    # function definition
    if args is None:
      args = getfullargspec(self.fn).args

    return tuple([
      self.fn.__module__,
      self.fn.__class__,
      self.fn.__name__,
      len(args or []),
    ])
```

In the snippet above, the `key` function returns a tuple that uniquely identifies the function in the codebase and holds

- the module of the function
- class to which the function belongs
- name of the function
- number of arguments the function accepts

The overridden `__call__` method invokes the wrapped function and returns the computed value (nothing fancy here right now). This makes the instance callable just like the function and it behaves exactly like the wrapped function.

```
def area(l, b):
  return l * b

>>> func = Function(area)
>>> func.key()
('__main__', <class 'function'>, 'area', 2)
>>> func(3, 4)
12
```

In the example above, the function `area` is wrapped in `Function` instantiated in `func`. The `key()` returns the tuple whose first element is the module name `__main__`, second is the class `<class 'function'>`, the third is the function name `area` while the fourth is the number of arguments that function `area` accepts which is `2`.

The example also shows how we could just call the instance `func`, just like the usual `area` function, with arguments `3` and `4` and get the response `12`, which is exactly what we'd get is we would have called `area(3, 4)`. This behavior would come in handy in the later stage when we play with decorators.

Building the virtual Namespace

Virtual Namespace, we build here, will store all the functions we gather during the definition phase. As there be only one namespace/registry we create a singleton class that holds the functions in a dictionary whose key will not be just a function name but the tuple we get from the `key` function, which contains elements that uniquely identify function in the entire codebase. Through this, we will be able to hold functions in the registry even if they have the same name (but different arguments) and thus facilitating function overloading.

```
class Namespace(object):
  """Namespace is the singleton class that is responsible
  for holding all the functions.
  """
  __instance = None

  def __init__(self):
    if self.__instance is None:
      self.function_map = dict()
      Namespace.__instance = self
    else:
      raise Exception("cannot instantiate a virtual Namespace again")

  @staticmethod
  def get_instance():
    if Namespace.__instance is None:
      Namespace()
    return Namespace.__instance

  def register(self, fn):
    """registers the function in the virtual namespace and returns
    an instance of callable Function that wraps the
    function fn.
    """
    func = Function(fn)
    self.function_map[func.key()] = fn
    return func
```

The `Namespace` has a method `register` that takes function `fn` as an argument, creates a unique key for it, stores it in the dictionary and returns `fn` wrapped within an instance of `Function`. This means the return value from the `register` function is also callable and (till now) its behavior is exactly the same as the wrapped function `fn`.

```
def area(l, b):
  return l * b

>>> namespace = Namespace.get_instance()
>>> func = namespace.register(area)
>>> func(3, 4)
12
```

Now that we have defined a virtual namespace with an ability to register a function, we need a hook that gets called during function definition; and here use Python decorators. In Python, a decorator wraps a function and allows us to add new functionality to an existing function without modifying its structure. A decorator accepts the wrapped function `fn` as an argument and returns another function that gets invoked instead. This function accepts `args` and `kwargs` passed during function invocation and returns the value.

A sample decorator that times execution of a function is demonstrated below

```
import time


def my_decorator(fn):
  """my_decorator is a custom decorator that wraps any function
  and prints on stdout the time for execution.
  """
  def wrapper_function(*args, **kwargs):
    start_time = time.time()

    # invoking the wrapped function and getting the return value.
    value = fn(*args, **kwargs)
    print("the function execution took:", time.time() - start_time, "seconds")

    # returning the value got after invoking the wrapped function
    return value

  return wrapper_function


@my_decorator
def area(l, b):
  return l * b


>>> area(3, 4)
the function execution took: 9.5367431640625e-07 seconds
12
```

In the example above we define a decorator named `my_decorator` that wraps function `area` and prints on `stdout` the time it took for the execution.

The decorator function `my_decorator` is called every time (so that it wraps the decorated function and s

By using Codementor, you agree to our Cookie Policy.

ace) the

Enjoy this post?

11

4

function in our virtual namespace. Hence we create our decorator named `overload` which registers the function in virtual namespace and returns a callable to be invoked.

```
def overload(fn):
  """overload is the decorator that wraps the function
  and returns a callable object of type Function.
  """
  return Namespace.get_instance().register(fn)
```

The `overload` decorator returns an instance of `Function` , as returned by `.register()` the function of the namespace. Now whenever the function (decorated by `overload`) is called, it invokes the function returned by the `.register()` function - an instance of `Function` and the `__call__` method gets executed with specified `args` and `kwargs` passed during invocation. Now what remains is implementing the `__call__` method in class `Function` such that it invokes the appropriate function given the arguments passed during invocation.

Finding the right function from the namespace

The scope of disambiguation, apart from the usuals module class and name, is the number of arguments the function accepts and hence we define a method called `get` in our virtual namespace that accepts the function from the python's namespace (will be the last definition for the same name - as we did not alter the default behavior of Python's namespace) and the arguments passed during invocation (our disambiguation factor) and returns the disambiguated function to be invoked.

The role of this `get` function is to decide which implementation of a function (if overloaded) is to be invoked. The process of getting the appropriate function is pretty simple - from the function and the arguments create the unique key using `key` function (as was done while registering) and see if it exists in the function registry; if it does then fetch the implementation stored against it.

```
def get(self, fn, *args):
  """get returns the matching function from the virtual namespace.

  return None if it did not fund any matching function.
  """
  func = Function(fn)
  return self.function_map.get(func.key(args=args))
```

The get function creates an instance of Function just so that it could use the key function to get a unique key and not replicate the logic. The key is then used to fetch the appropriate function from the function registry.

Invoking the function

As stated above, the __call__ method within class Function is invoked every time a function decorated with an overload decorator is called. We use this function to fetch the appropriate function using the get function of namespace and invoke the required implementation of the overloaded function. The __call__ method is implemented as follows

```
def __call__(self, *args, **kwargs):
  """Overriding the __call__ function which makes the
  instance callable.
  """
  # fetching the function to be invoked from the virtual namespace
  # through the arguments.
  fn = Namespace.get_instance().get(self.fn, *args)
  if not fn:
    raise Exception("no matching function found.")

  # invoking the wrapped function and returning the value.
  return fn(*args, **kwargs)
```

The method fetches the appropriate function from the virtual namespace and if it did not find any function it raises an Exception and if it does, it invokes that function and returns the value.

Function overloading in action

Once all the code is put into place we define two functions named area : one calculates the area of a rectangle and the other calculate the area of a circle. Both functions are defined below and decorated with an overload decorator.

```
@overload
def area(l, b):
  return l * b

@overload
def area(r):
  import math
  return math.pi * r ** 2


>>> area(3, 4)
12
>>> area(7)
153.93804002589985
```

When we invoke `area` with one argument it returns the area of a circle and when we pass two arguments it invokes the function that computes the area of a rectangle thus overloading the function `area` . You can find the entire working demo here.

Conclusion

Python does not support function overloading but by using common language constructs we hacked a solution to it. We used decorators and a user-maintained namespace to overload functions and used the number of arguments as a disambiguation factor. We could also use data types (defined in decorator) of arguments for disambiguation - which allows functions with the same number of arguments but different types to overload. The granularity of overload is only limited by function `getfullargspec` and our imagination. A neater, cleaner and more efficient approach is also possible with the above constructs so feel free to implement one and tweet me @arpit_bhayani, I will be thrilled to learn what you have done with it.

This article was originally published on my blog - Overload Functions in Python.

If you liked what you read, subscribe to my newsletter and get the post delivered directly to your inbox and give me a shout-out @arpit_bhayani.

Python

Enjoy this post? Give **Arpit Bhayani** a like if it's helpful.

11 4 SHARE

Enjoy this post?

11 4

Backend @Unacademy • Data @Amazon • Platform @Practo | Writes about Language internals and Math in Computer Science

I'm an avid programmer, passionate about code, design, startups and technology. Currently I am working at Amazon as Software Development Engineer 2. Before Amazon I was working for a healthcare startup named Practo where I single ...

FOLLOW

4 Replies

Leave a reply

yotamolenik a year ago

maybe its more of a philosophical question - but why doesn't Python have overloading since the beginning? what is the downside?

Reply

Daniel Riggs 2 years ago

As of python 3.8, an overload decorator is available in the standard library: https://docs.python.org/3/library/typing.html#typing.overload

There is also functools.singledispatch which can do the same thing: https://docs.python.org/3/library/functools.html

Show more

1 Reply

jyothi kiran 3 years ago

Sir, i want your email id.

Reply

Show more replies

Enjoy this post?

11 4