doing-without-class

October 3, 2021

1 Doing without `class`

A talk by Tibs (they/he), for CamPUG, on Tuesday 5th October 2021

Sources are at https://github.com/tibs/doing-without-class

Presented using Jupyter Notebook

Licensed under a Creative Commons Attribution-ShareAlike 4.0 International License **except** for the flowchart image from https://terribleminds.com/ramble/2013/08/06/are-you-a-real-writer-a-handy-and-hasty-flowchart/ which belongs to Chuck Wendig.

1.1 Abstract

When I named this talk, I realised that the name was ambiguous.

I could have changed the name, but instead I figured I'd do both talks.

- Is it OK to write Python code without classes?
- Can we create Python classes without using the `class` keyword

1.2 Part 1: Is it OK to write Python code without classes?

tldr; Yes

Ideas:

- Write the simplest thing that will get the job done
- Python is deliberately a multi-paradigm language

The seven programming ur-languages by Fred Ross, 2021

Broadly: ALGOL, Lisp, ML, Self, Forth, APL, Prolog

Python clearly fits well in the "ALGOL" family.

> **Characteristics**. Programs consist of sequences of assignments, conditionals, and loops, organized into functions. Many languages add module systems, ways of defining new data types, polymorphism, or alternate control flow constructs like exceptions or coroutines.
>
> Most common programming languages trace to this ur-language

Stop Writing Classes

A talk from PyCon US 2012, by Jack Diederich

> Classes are great but they are also overused. This talk will describe examples of class overuse taken from real world code and refactor the unnecessary classes, exceptions, and modules out of them."

There's nothing wrong with:

1. Write some simple linear code
2. Ooh - functions would make this easier - refactor to use functions
3. Ooh - maybe I need a class - refactor to use it

Some history ...from a C programmer

Back in the late 1980s, we were writing C code that created a `context` datastructure, which we would pass to functions.

This would contain the bundle of data items that most functions needed most of the time.

We would put it as the first argument to those functions.

Later on, we had the idea to add pointers to common functions to those contexts.

So we'd typically have `print` and `compare` function pointers, and we'd attach the appropriate function *for that particular type of context.*

Which we'd do in an `init` function.

Yes, we too invented some of the basics of OO.

Just like lots of people, I'm sure.

What's the point?

Well, sometimes it's OK to pass a context value around, without turning things "inside out" and creating a class.

And, to be honest, if there are only a few values in that context, it may not even be worth creating it. Just pass the values.

Some history ...from a Fortan programmer

Before we used C, we used Fortran. It was somewhere between FORTRAN IV and Fortran 77.

So a lot of the communication between functions and subroutines was done with `COMMON` blocks - essentially named global storage.

We managed.

Personally, I am a library writer at heart, so am unreasonably prejudiced against using Python `global`s.

But if you're writing a simple *program*

- and all of the functions in that program use a few shared values,
- and especially if few (if any) of them modify those values

then globals can make sense.

In summary

- It's OK to keep things simple
- It's OK not to use classes if they're not needed
- It's OK to pass around `context` values
- It's OK (in a program, not in a library!) to use `global` values

But be prepared to refactor when it gets hard to understand.

Addendum I think people sometimes ask questions like this because they're not sure if they're "a real programmer" if they don't tick certain boxes / do certain things.

I think Chuck Wendig's flowchart for "Are you a real writer" is particularly useful - just change "writer" to be "programmer".

1.3 Part 2: Can we create Python classes without using the `class` keyword

(and some other stuff)

Summary:

- Looking at a simple class
- Can we define a class without class? (*tldr;* yes)
- Can we construct an instance by hand? (*tldr;* yes-ish)
- Can we create a function without using def? (*tldr;* sort-of no)

1.3.1 Looking at a simple class

[1]:
```
class Example:
    """A simple example class"""
    a = 3
    def f(self, p):
        """A function that takes a single argument"""
        return f'The parameters were {self} and {p}'
```

Which we can use in the expected manner

[2]:
```
e = Example()
print(f'Class {Example}')
print(f'Instance {e}')
print(f'Value    {e.a}')
print(f'Method   {e.f}')
print(f'Calling the method with 3 {repr(e.f(3))}')
```

```
Class <class '__main__.Example'>
Instance <__main__.Example object at 0x103da08b0>
Value    3
Method   <bound method Example.f of <__main__.Example object at 0x103da08b0>>
Calling the method with 3 'The parameters were <__main__.Example object at
0x103da08b0> and 3'
```

1.3.2 Using values

We can update the `a` on the class and see it also change on the instance

[3]:
```
Example.a = 4
print(Example.a)
print(e.a)
```

```
4
4
```

But if we update the `a` on the instance

[4]:
```
print(e.a)
e.a += 1
print(e.a)
```

```
4
5
```

It does not change `a` on the class

[5]:
```
print(Example.a)
```

```
4
```

And now changing it on the class won't touch the value on the instance

[6]:
```
Example.a = 99
print(f'Class    "a" {Example.a}')
print(f'Instance "a" {e.a}')
```

```
Class    "a" 99
Instance "a" 5
```

If we add a new value to the class, the instance will get it

[7]:
```
Example.b = 12
print(f'Class    "b" {Example.b}')
print(f'Instance "b" {e.b}')
```

```
Class    "b" 12
Instance "b" 12
```

But not the other way round

[8]:
```
e.c = -1
print(f'Instance has "c" {hasattr(e, "c")}')
print(f'Instance "c" {e.c}')
print(f'Class has "c" {hasattr(Example, "c")}')
try:
    print(f'Class    "c" {Example.c}')
```

```
except Exception as exc:
    print(f'{exc.__class__}: {exc}')
```

```
Instance has "c" True
Instance "c" -1
Class has "c" False
<class 'AttributeError'>: type object 'Example' has no attribute 'c'
```

Aside Why code that as

```
try:
    <thing>
except Exception as exc:
    print(f'{exc.__class__}: {exc}')
```

rather than let the exception "just happen"?

Purely pragmatism - I expect the exception, and if I don't catch it we get to see the traceback (which is generally a Good Thing) but if I try to run all of the notebook using "Cells / Run All", it will stop at the first traceback, which is inconvenient when testing the talk.

1.3.3 Using methods

We can call the method on the instance:

[9]:
```
e.f(4)
```

[9]:
```
'The parameters were <__main__.Example object at 0x103da08b0> and 4'
```

Or via the class, although then we need to pass in the instance (`self`) explicitly

[10]:
```
Example.f(e, 4)
```

[10]:
```
'The parameters were <__main__.Example object at 0x103da08b0> and 4'
```

and there's nothing special about `self`

[11]:
```
Example.f('not an instance', 4)
```

[11]:
```
'The parameters were not an instance and 4'
```

So we shouldn't be surprised that on the class it's a function,

but on the instance it's a method,

which gets passed the instance as its first argument

[12]:
```
print(f'Example.f is a {type(Example.f)}')
print(f'e.f is a {type(e.f)}')
```

```
Example.f is a <class 'function'>
e.f is a <class 'method'>
```

Aside In Python, all methods are functions, and there's nothing special about the name `self`.

[13]:
```
def double(self, x): return x + x

Example.double = double
print(f'Class "Example" has "double" {hasattr(Example, "double")}')
print(f'Example.double is a {type(Example.double)}')
print(f'Example.double("thing", 3) is {Example.double("thing", 3)}')

print(f'Instance "e" has "double" {hasattr(e, "double")}')
print(f'e.double is a {type(e.double)}')
print(f'e.double(3) is {e.double(3)}')
```

```
Class "Example" has "double" True
Example.double is a <class 'function'>
Example.double("thing", 3) is 6
Instance "e" has "double" True
e.double is a <class 'method'>
e.double(3) is 6
```

Contrariwise, in Ruby, all functions are methods:

```
irb(main):005:1* def fn(a)
irb(main):006:1*   puts "self is #{self} #{self.inspect}"
irb(main):007:0> end
=> :fn
irb(main):008:0> fn(1)
self is main main
=> nil
```

It seems natural to me that in Python we have to specify the `self` argument explicitly, as it's not special *to the function.*

However, in Ruby, as all functions are methods, there's always a parent class, and always a `self`, so there's no need to put it in the argument list.

End of aside

1.3.4 Can we define a class without `class`?

Can we create a class without using the `class` keyword?

We've already used the `type` callable:

[14]:
```
type(Example)
```

[14]:
```
type
```

(Although I would have guessed a class would be of type "class")

and of course

[15]:
```
print(type(1))
print(type('2'))
```

```
<class 'int'>
<class 'str'>
```

While `type` looks like a function, interestingly it isn't

If we ask for `help(type)`, we see:

```
Help on class type in module builtins:

class type(object)
 |  type(object_or_name, bases, dict)
 |  type(object) -> the object's type
 |  type(name, bases, dict) -> a new type
 ...
```

We've been using the first of those

`type(object) -> the object's type`.

and now we want the second

`type(name, bases, dict) -> a new type`

[16]:
```
EmptyClass = type('EmptyClass', (), {})
```

[17]:
```
print(type(EmptyClass))
print(repr(EmptyClass))
```

```
<class 'type'>
<class '__main__.EmptyClass'>
```

[18]:
```
ec = EmptyClass()
print(type(ec))
print(repr(ec))
```

```
<class '__main__.EmptyClass'>
<__main__.EmptyClass object at 0x103dd61c0>
```

The middle argument specifies base classes

[19]:
```
AnInteger = type('AnInteger', (int,), {})
x = AnInteger()
print(x)
```

```
0
```

The last argument can be used to set values on the new class

[20]:
```python
ClassWithValues = type('ClassWithValues', (), {'a': 99, 'f': lambda self: 1})
cv = ClassWithValues()
print(cv.a)
print(cv.f())
```

```
99
1
```

And we can use both if we want

[21]:
```python
def double(x): return x * 2
DoublingInteger = type('DoublingInteger', (int,), {'double': double})
db = DoublingInteger(9)
print(db)
print(db.double())
```

```
9
18
```

Let's build a simple class

[22]:
```python
def function_f(self, p):
    """A function we shall use as a method that takes a single argument"""
    return f'The parameters were {self} and {p}'
```

[23]:
```python
ByHand = type('ByHand', (), {'f': function_f, 'a': 3})
```

[24]:
```python
bh = ByHand()
print(bh)
print(bh.a)
print(bh.f(2))
```

```
<__main__.ByHand object at 0x103df12b0>
3
The parameters were <__main__.ByHand object at 0x103df12b0> and 2
```

The obvious next thing to do is to make a function to make classes

[25]:
```python
from collections import ChainMap
def make_a_class(name, value_dict, method_dict):
    cls = type(name, (), dict(ChainMap(value_dict, method_dict)))
    return cls
```

[26]:
```python
C = make_a_class('ByHand', {'a': 3}, {'f': function_f})
print(f'Class {C!r}')
print(f'Class value a {C.a!r}')
print(f'Class function {C.f(None, "fred")}')
```

```
Class <class '__main__.ByHand'>
Class value a 3
Class function The parameters were None and fred
```

We can create an instance, just as we might expect

[27]:
```
o = C()
print(f'Instance {o!r}')
print(f'Instance vaue a {o.a!r}')
print(f'Instance function {o.f("fred")}')
```

```
Instance <__main__.ByHand object at 0x103df1d00>
Instance vaue a 3
Instance function The parameters were <__main__.ByHand object at 0x103df1d00>
and fred
```

1.3.5 Can we construct an instance by hand?

Can we create an empty object and add things to it?

Our first guess might be to create an instance of the base class, `Object`

[28]:
```
o = object()
```

but unfortunately, its not possible to add new values to instances of `Object`

[29]:
```
try:
    o.a = 1
except Exception as exc:
    print(f'{exc.__class__}: {exc}')
```

```
<class 'AttributeError'>: 'object' object has no attribute 'a'
```

So we still have to use `type` to get an empty mutable object

[30]:
```
EmptyClass = type('EmptyClass', (), {})
eo = EmptyClass()
print(type(eo))
```

```
<class '__main__.EmptyClass'>
```

And we know we can do

[31]:
```
eo.a = 1
print(eo.a)
```

```
1
```

[32]:
```
eo.a = eo.a + 1
print(eo.a)
```

```
2
```

Can we add a function *to the object* and have it be a method?

[33]:
```
def maybe_a_method(self, x):
    print(f'Maybe a method on {self} and {x}')

eo.f = maybe_a_method
try:
    print(eo.f(1))
except Exception as exc:
    print(f'{exc.__class__}: {exc}')
```

```
<class 'TypeError'>: maybe_a_method() missing 1 required positional argument:
'x'
```

Unfortunately, adding a function as a value on an instance doesn't make a method

[34]:
```
print(type(eo.f))
```

```
<class 'function'>
```

Normally, when we ask an instance for a method (`eo.f`), it gets looked up in the instance, isn't found there, and is looked up in the class, which says "I know what you're doing, that's a function you're looking up on me, so you must want a method back"

[35]:
```
class NoMethods: pass
def just_a_function(self): return 'Aha!'
NoMethods.f = just_a_function
nm = NoMethods()

print(type(just_a_function))
print(type(NoMethods.f))
print(type(nm.f))
print(nm.f())
```

```
<class 'function'>
<class 'function'>
<class 'method'>
Aha!
```

But our empty object is an instance of an empty class, so that won't work.

Luckily there is a way:

[36]:
```
eo.f = just_a_function.__get__(eo, EmptyClass)
print(type(eo.f))
print(eo.f())
```

```
<class 'method'>
Aha!
```

I don't propose to explain that (but am grateful to https://stackoverflow.com/a/46757134 for the example!).

If you want to learn more, then this is using the power of *descriptors*, which are at the heart of Python's attribute access

See the HOWTO at https://docs.python.org/3/howto/descriptor.html

There is also a more "understandable" way to do this.

We can create a `method` from our function

[37]:
```
import types
eo.f = types.MethodType(just_a_function, eo)
print(type(eo.f))
print(eo.f())
```

```
<class 'method'>
Aha!
```

And we can create a function to wrap this nicely

[38]:
```
def pretend_instance(class_name, variable_dict, function_list):
    eo_class = type(class_name, (), variable_dict)
    eo = eo_class()
    for f in function_list:
        setattr(eo, f.__name__, types.MethodType(f, eo))
    return eo
```

[39]:
```
x = pretend_instance('ClassName', {'var': 3}, [just_a_function])
print(type(x))
print(x.var)
print(x.just_a_function())
```

```
<class '__main__.ClassName'>
3
Aha!
```

1.3.6 Can we create a function without using `def`?

Well, there is lambda

[40]:
```
lamb = lambda x: x + 1

lamb(2)
```

[40]:
```
3
```

although

1. That's another keyword
2. It's very limited in what it can do

   An anonymous inline function consisting of a single expression which is evaluated when the function is called

There is also `types.FunctionType`, which is similar in idea to our use of `type` to create classes.

(unfortunately, its signature is implementation specific and may even change between Python versions)

```
>>> help(types.FunctionType)
Help on class function in module builtins:

class function(object)
 |  function(code, globals, name=None, argdefs=None,
 |           closure=None)
 |
 |  Create a function object.
 |
 |  code
 |    a code object
 |  globals
 |    the globals dictionary
 ...
```

We can get a code object from an existing function

[41]:
```
print(lamb.__code__)
```

```
<code object <lambda> at 0x103ddc870, file "/var/folders/3c/74720f_907b07qy3vwck
lsg00000gp/T/ipykernel_88426/2561191998.py", line 1>
```

[42]:
```
import types
lambish = types.FunctionType(lamb.__code__, globals())
print(lambish(1))
```

```
2
```

or we can create one with `compile`

[43]:
```
code = compile('print(4)', 'no-file', 'exec')
compiled_fn = types.FunctionType(code, globals())
print(compiled_fn())
```

```
4
None
```

But how did `lambish` know about its argument?

[44]:
```
print(dir(lamb.__code__))
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'co_argcount', 'co_cellvars', 'co_code', 'co_consts',
'co_filename', 'co_firstlineno', 'co_flags', 'co_freevars', 'co_kwonlyargcount',
'co_lnotab', 'co_name', 'co_names', 'co_nlocals', 'co_posonlyargcount',
'co_stacksize', 'co_varnames', 'replace']
```

The documentation for the `inspect` module tells us that `co_argcount` is the number of arguments, and `co_varnames` is a tuple of the names of the arguments and then the names of the local variables

[45]:
```
print(lamb.__code__.co_argcount)
print(lamb.__code__.co_varnames)
```

```
1
('x',)
```

But that's not mutable

[46]:
```
try:
    lamb.__code__.co_varnames = ('x', 'y')
except Exception as exc:
    print(f'{exc.__class__}: {exc}')
```

```
<class 'AttributeError'>: readonly attribute
```

And at this point, I haven't been able to find how to define the arguments for a `code` value.

1.3.7 Some other minor amusements

Looking at bytecode We can look at the bytecode for a callable (https://docs.python.org/3/library/dis.html)

[47]:
```
import dis
dis.dis(lambish)
```

```
  1           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (1)
              4 BINARY_ADD
              6 RETURN_VALUE
```

That should be fairly recognisable as our lambda function.

Unsurprisingly, what that did was look up the `__code__` value on `lambish` for us

[48]:
```
dis.dis(lamb.__code__)
```

```
  1           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (1)
              4 BINARY_ADD
              6 RETURN_VALUE
```

The `inspect` module The `inspect` module (https://docs.python.org/3/library/inspect.html) has many useful ways to introspect Python code. Here are a few

Loooking up the signature for a callable

[49]:
```
import inspect
print(inspect.signature(lambish))
```

```
(x)
```

Getting the source code for an object

[50]:
```
print(inspect.getsource(Example.f))
```

```
    def f(self, p):
        """A function that takes a single argument"""
        return f'The parameters were {self} and {p}'
```

Asking about the general kind of an object

[51]:
```
print(inspect.isfunction(just_a_function))
print(inspect.ismethod(just_a_function))
print(inspect.ismethod(e.f))
print(inspect.isclass(make_a_class('ClassThing', {'a': 3}, {})))
```

```
True
False
True
True
```

1.3.8 Some useful links

If you're interested in how Python works, then the book Fluent Python by Luciano Ramalho is excellent.

If you want to dig into how CPython is implemented, then CPython Internals: Your Guide to the Python 3 Interpreter by Anthony Shaw looks excellent (I've still to read it properly).

I found the following to be useful while researching this:

- https://stackoverflow.com/questions/19476816/creating-an-empty-object-in-python has interesting takes on creating an empty object
- https://stackoverflow.com/questions/13184281/python-dynamic-function-creation-with-custom-names taks about ways to use inline `def` to construct functions, and there's a link to https://smarie.github.io/python-makefun/, which seems to be a comprehensive solution for constructing functions at runtime (I've not used it)
- https://stackoverflow.com/questions/394770/override-a-method-at-instance-level has a discussion on why assigning a function to an instance doesn't set it on the class

If you're interested in `descriptors`, Raymond Hettinger's Descriptor HowTo Guide is excellent (I tend to recommend everything by him).

In the Python documentation, we've referenced, at least in passing:

- https://docs.python.org/3/library/types.html describes the `types` module, "Dynamic type creation and names for built-in types"

- https://docs.python.org/3/library/functions.html#object describes the `object` callable
- https://docs.python.org/3/library/functions.html#type describes the `type` callable
- https://docs.python.org/3/library/functions.html#compile describes the `compile` function, to produce byte code from source code
- https://docs.python.org/3/library/inspect.html describes the `inspect` module, which is worth a read just to see what you can find out about objects
- https://docs.python.org/3/library/dis.html describes the `dis` module, and how to disassemble callables - honestly, something you shouldn't *need* to do, but it can be interesting

1.3.9 Fin

A talk by Tibs (they/he), for CamPUG, on Tuesday 5th October 2021

Sources are at https://github.com/tibs/doing-without-class

Presented using Jupyter Notebook

Licensed under a Creative Commons Attribution-ShareAlike 4.0 International License **except** for the flowchart image from https://terribleminds.com/ramble/2013/08/06/are-you-a-real-writer-a-handy-and-hasty-flowchart/ which belongs to Chuck Wendig.

```
[ ]:
```