Python

metaprogramming related cool story

Aleksandr Koshkin
vk.com/magniff
github.com/magniff

type is THE __class__ of all classes

```
>>> klass = type(
...       'MyAwesomeClass',
...       (object,),
...       {
...           'foo': 'bar',
...           '__init__': lambda self: print('__init__ is called')
...       }
... )
>>> klass
0: <class '__main__.MyAwesomeClass'>
>>> klass()
__init__ is called
1: <__main__.MyAwesomeClass object at 0x7f0a54eaff98>
```

and an instance of class ‘object’… errrr

```
>>> isinstance(object, type)
0: True
>>> isinstance(type, object)
1: True
```

we can customize ‘type’ behaviour

```
class MetaClass(type):
    def __new__(mcls, name, bases, attrs, *args, **kwargs):
        # custom logic here
        do_magic_with(name, bases, attrs, *args, **kwargs)
        # delegate actual build to superclass (type)
        return super().__new__(mcls, name, bases, attrs, *args, **kwargs)


class MyAwesomeClass(metaclass=MetaClass):
    pass


# magic happens here...
```

disassemble class defenition

```python
from dis import dis
dis("class A: pass")
```

```
1              0 LOAD_BUILD_CLASS
               1 LOAD_CONST              0 (<code object A at ...)
               4 LOAD_CONST              1 ('A')
               7 MAKE_FUNCTION           0
              10 LOAD_CONST              1 ('A')
              13 CALL_FUNCTION           2 (2 positional, 0 keyword pair)
              16 STORE_NAME              0 (A)
              19 LOAD_CONST              2 (None)
              22 RETURN_VALUE
```

the heart of python

```
#Python/ceval.c
PyObject *
PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)

for (;;) {
    # tl;dr
    switch (opcode) {
        TARGET(NOP)
        TARGET(LOAD_CONST)
        ...
        TARGET(LOAD_BUILD_CLASS) {
            # skip some more ... duuuh
            bc = _PyDict_GetItemId(f->f_builtins, &PyId___build_class__);
            PUSH(bc);
        }
}
```

__builtins__ are implemented in C

```
#Python/bltinmodule.c
```

```
static PyObject *
builtin___build_class__(PyObject *self, PyObject *args, PyObject *kwds)
{
    # skip long foreplay
    cls = PyEval_CallObjectWithKeywords(meta, margs, mkw);
    return cls;
}
```

load_build_class → **__build_class__** → PyEval_CallObjectWithKeywords

little __builtins__ hack

```
>>> bb = __builtins__.copy()
... class MyBI:
...   def __init__(self, bb):
...     self.bb = bb
...   def __getitem__(self, item):
...     if item.startswith('magic_'):
...       bb['print']('hello from magic!')
...   def __setitem__(self, item):
...     #some set logic
...     pass
...
... __builtins__ = MyBI(bb)
>>> magic_some_var
hello from magic!
```

wrap magic part with context manager

```
class _ContextEntry:

    import builtins
    context = _ContextInternal(builtins_module=builtins)

    def __init__(self, klass_builder):
        self.context._register_klass_builder(klass_builder)

    def __enter__(self):
        self.context.enable()

    def __exit__(self, klass, value, tb):
        self.context.disable()
```

and the frontend is

```python
from magic import wonderland


def callback(builder, *args, **kwargs):
    return builder(*args, **kwargs)


with wonderland(callback):
    class A(foo=bar, some=int, metaclass=MyMeta) :
        pass
```

and the frontend is

```
def callback(builder, *args, **kwargs):
    func, name, *bases = args
    print(name)
    return builder(*args, **kwargs)
```

```
def callback(builder, *args, **kwargs):
    func, name, *bases = args
    print(func.__code__.co_consts)
    return builder(*args, **kwargs)
```

… and so forth

so, all the class data been handled by func.__code__ object

what makes any code better?

what makes any code better?

metaclasses, ofc

bind metaclass by automagic

```python
from magic import wonderland


class Meta(type):
    def __new__(cls, name, bases, attrs):
        return super().__new__(cls, name, bases, attrs)


def callback(builder, *args, **kwargs):
    kwargs['metaclass'] = Meta
    return builder(*args, **kwargs)


with wonderland(callback):
    class A: pass
```

example: simple logger meta

```python
class LoggerMeta(type):

    @classmethod
    def _patch_method(cls, method):
        def new_method(*args, **kwargs):
            print(
                "call method '{method}' with\n"
                "args: {args}\nkwargs: {kwargs}".format(
                    args=args, kwargs=kwargs, method=method.__name__
                )
            )
            return method(*args, **kwargs)

        return new_method

    def __new__(cls, name, bases, attrs):
        for attr_name, attr_value in attrs.items():
            if not isinstance(attr_value, FunctionType):
                continue
            attrs[attr_name] = cls._patch_method(attr_value)

        return super().__new__(cls, name, bases, attrs)
```

but real world is cruel

```
class SomeMetaMeta:
    # some meta meta, why not?
    pass

class MyMetaclass(metaclass=SomeMetaMeta):
    # some meta
    pass

class MyClass(metaclass=MyMetaclass):
    # actual class
    pass


@some_klass_decorator(var0, var1)
class AwesomeClass(MyClass, metaclass=SomeMoreMeta):
    # easy man!!1
    pass
```

so problems summary

- metaclass conflict
- decorator re-apply
- non clean metas
- weird __import__ related behaviour
- to much magic per line
- lots of them

so problems summary

. metaclass conflict

. decorator re-apply

. non clean metas

. weird __import__ related behaviour

. to much magic per line

. lots of them

would I recommend it in production code? srsly?

so problems summary

. metaclass conflict

. decorator re-apply

. non clean metas

. weird __import__ related behaviour

. to much magic per line

. lots of them

would I recommend it in production code? srsly?

did I get some fun? you bet!!!1