Building DSLs in Python

A toolbox of techniques

Why Python?

- Dynamically typed
- First class support for reflection
  - is(instance|subclass), (has|get|set|del)attr
- Flexible object model
- Metaprogramming
- **Everyone knows it**

Python is Slow

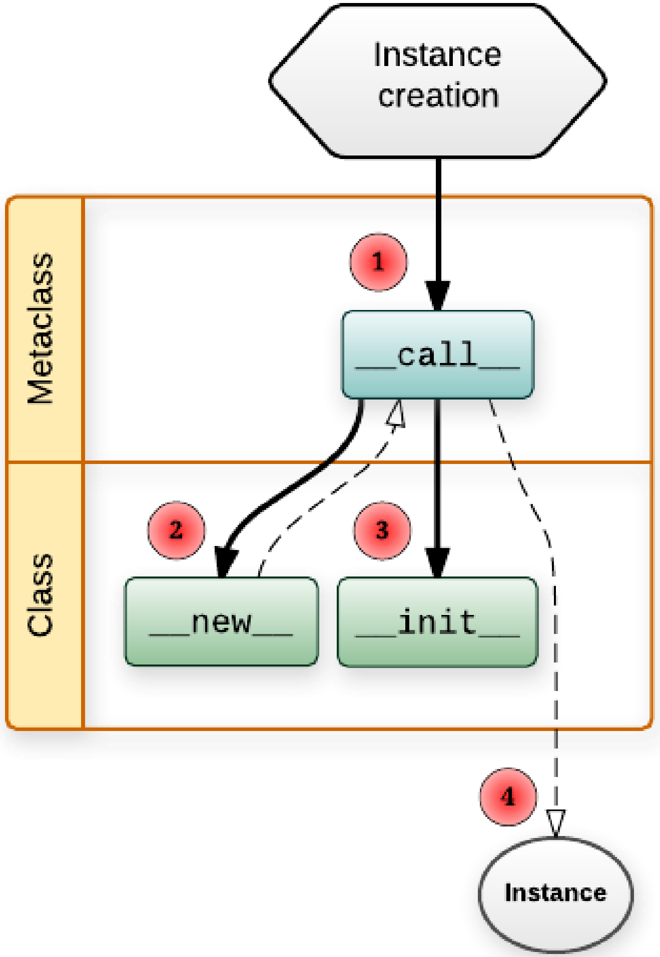
- Doesn’t matter
  - The performance of host language is performance of compiler
- Can be used to program fast c libraries.

Python has a very rigid execution model

But it has a lot of call backs

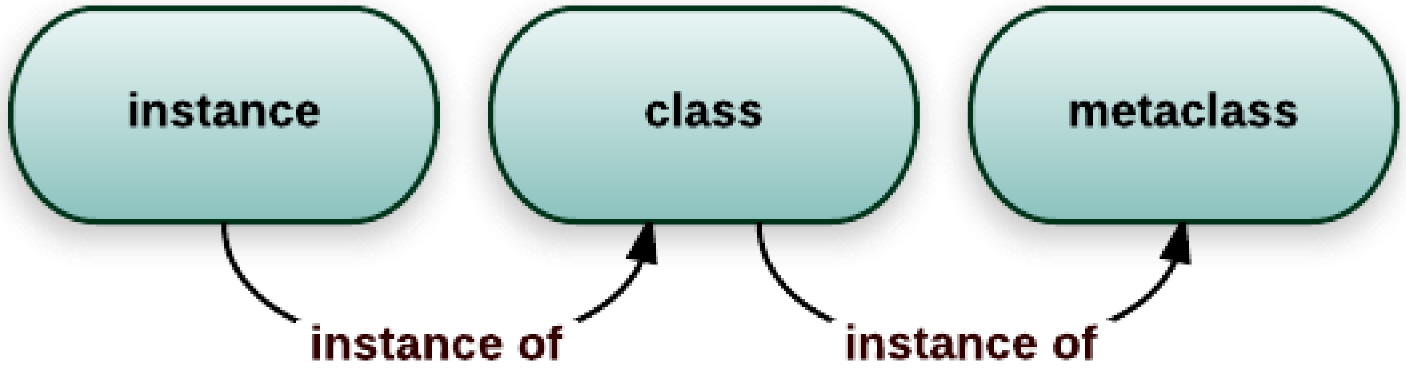
Understanding Object Creation

```
class Foo:          ???
    pass

--------------      ----------------------------

Foo()               type.__call__(cls=Foo):
                        #type == type(cls)
                        obj = cls.__new__(cls)
                        if isinstance(obj, cls):
                            obj.__init__()
                        return obj
```



https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/

Wait whats a metaclass?



https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/

Two constructors? This is madness!

```
__new__  ::
   class -> args -> instance
```

- Constructor
- Controls object creation
- Generally unnecessary

```
__init__ ::
  instance -> args -> None
```

- Initializer
- Controls object initialization
- Probably want to write one of these

__new__ what is it good for?

```
class Singleton:
    _instance = None
    def __new__(cls):
        return None

    @classmethod
    def get_instance(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            if isinstance(cls._instance, cls):
                cls._instance.__init__()
        return cls._instance
```

```
class Highlander(Signleton):
    def __init__(self):
        ...

assert Highlander() == None
obj = Highlander.get_instance()
assert obj != None
```

But who cares if init runs twice?

```
class Singleton:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            obj = super().__new__(cls)
            cls._instance = obj
        return cls._instance
```

But who cares if init runs twice?

```
class Singleton:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            obj = super().__new__(cls)
            cls._instance = obj
        return cls._instance
```

```
class UniqueResource(Singleton):
    def __init__(self):
        # Aquire unique resource
```

But who cares if init runs twice?

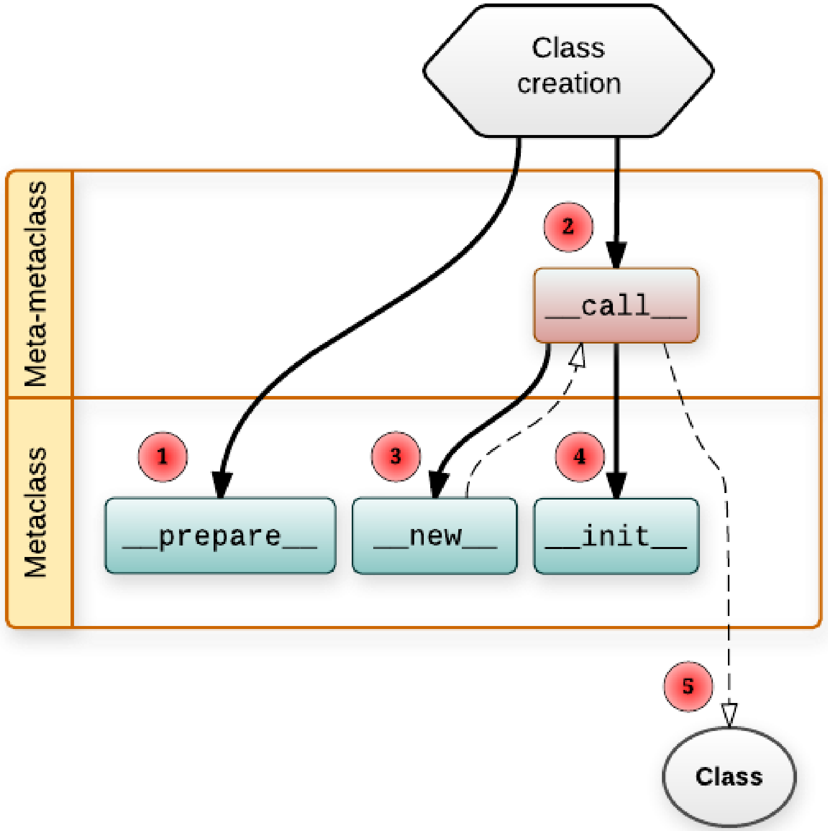
```
class Singleton:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            obj = super().__new__(cls)
            cls._instance = obj
        return cls._instance
```

```
class UniqueResource(Singleton):
    def __init__(self):
        if hasattr(self, "init_done"):
            return
        # Aquire unique resource
        self.init_done = True
```

Understanding Class Creation

```python
class FooMeta(type): ...

class Foo(metaclass=FooMeta):
    x = 1
    def __init__(self):
        ...
```



https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/

Understanding Class Creation

```
ns = FooMeta.__prepare__(
    "Foo", # name
    (),  # base classes
)

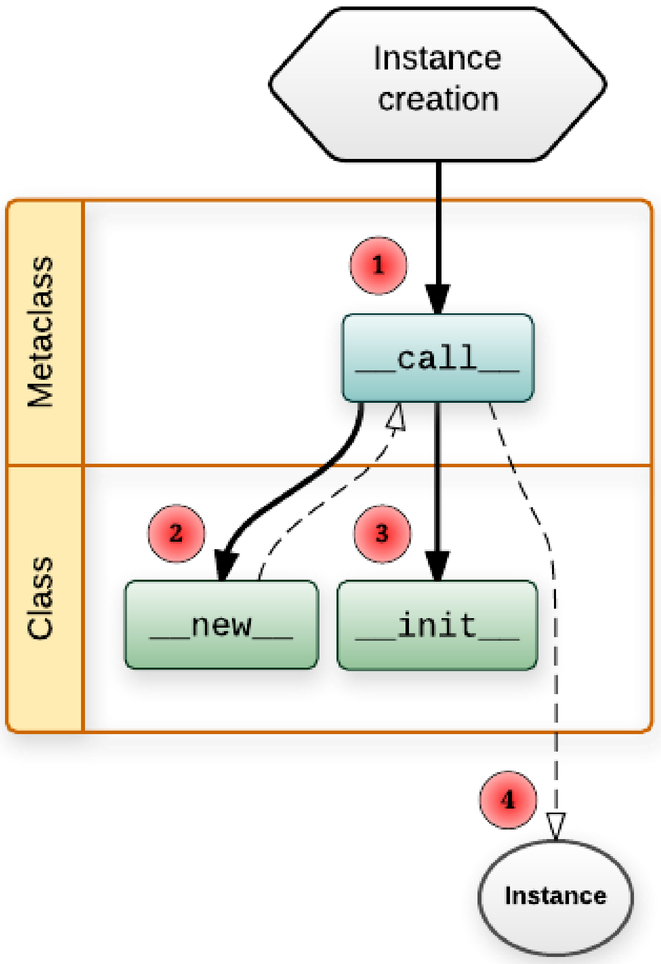
body = """\
x = 1
def __init__(self):
    ...
"""

exec(body, globals(), ns)
# ns = {
#   'x': 1,
#   '__init__': <function>
# }
```

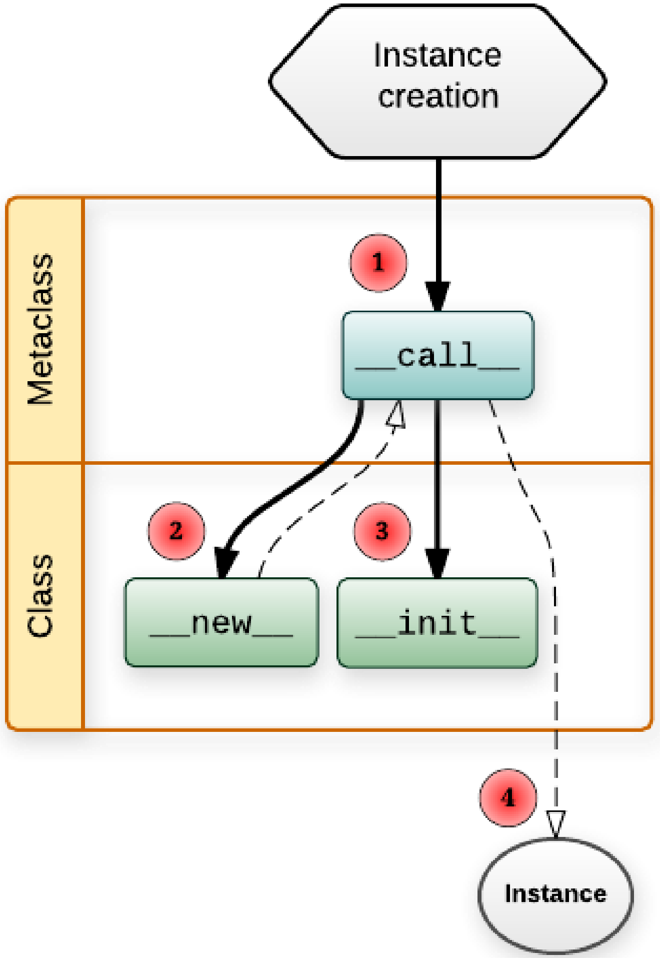
```
# type(FooMeta) == type
# i.e. FooMeta's metaclass is type
Foo = type.__call__(
        mcs=FooMeta,
        name="Foo",
        bases=(),
        namespace=ns):

    cls = mcs.__new__(
        mcs,
        name,
        bases,
        namespace
    )
    if isinstance(cls, mcs):
        cls.__init__(name, bases, namespace)
    return cls
```

Singleton with Metaclasses

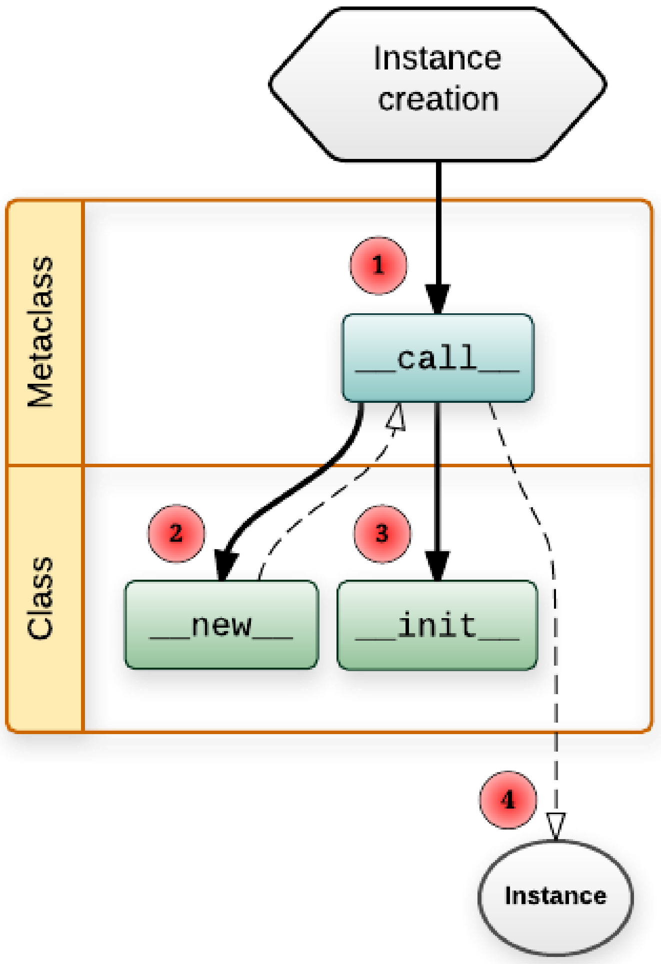


Singleton with Metaclasses




```
class SingletonMeta:
    _cache = {}
    def __call__(cls):
        mcs = type(cls)
        if cls in mcs._cache:
            return mcs._cache[cls]
        obj = super().__call__()
        mcs._cache[cls] = obj
        return obj
```

Singleton with Metaclasses




```
class SingletonMeta:
    _cache = {}
    def __call__(cls):
        mcs = type(cls)
        try:
            return mcs._cache[cls]
        except KeyError:
            pass
        obj = super().__call__()
        mcs._cache[cls] = obj
        return opj
```

I am going to use metaclasses for everything!

- Every class can only have one metaclass
  - Makes it difficult to inherit
    - If you want to use your own
- Use metaclasses where necessary
  - But avoid their use when possible

Building parameterized types

- Init argument
  - numpy
- Class Attributes
  - enum
- Class Anotations
  - dataclasses
- Getitem
  - typing

```
np.array(vals) # shape is inferred from vals
np.empty(shape)

class Flags(Enum):
    a = 1
    b = 2

@dataclass
class Point:
    x: int
    y: int

T = Union[int, str]
```

Class Caching

- Same parameterization should generate the same class
  - `T[args] is T[args]`
  - ```
    class S(T): param = args
    class V(T): param = args
    S == V #(or potentially S is V)
    ```
- isinstance and issubclass
  - Class Attributes / Anotations
    - Must be done in metaclass or decorator
  - Getitem
    - May be done in baseclass or metaclass
    ```

Building a fixed sized array class

3.5 versions

Code Example

Using init arguments

- Upside:
  - No metaclass necessary
  - Simple
- Downside:
  - Does not work with isinstance or issubclass
  - All arrays are the same python type

Attribute classes

- Upside:
  - Works with isinstance or issubclass
  - Works with inheritence
- Downside:
  - Require metaclass or decorator for caching and magic inheritance
    - Could validate type with __init_subclass__
  - Requires a name

GetItem classes

- Upside:
  - Works with isinstance or issubclass
  - No naming required
- Downside:
  - May have strange interactions with inheritence
    - Metaclass required to fix issues

Escaping python’s execution model

AST rewriting

- `import ast`
  - Unstable – version specific
  - Can’t round trip code
  - mutable
- astor – utilities
- green tree snake – documentation

libcst

- Version independent
  - 3.0 to 3.8 (inclusive)
- Keeps all details
  - Can round trip code
- Immutable
- Strongly typed
- Sometimes too verbose

AST Tools

- A library for rewriting CST
- Allows decorators to rewrite functions and class definitions
- Pass infrastructure
    - Easy chaining of transformers
- https://github.com/leonardt/ast_tools

Visitors: more than meets the eye

- Provides an interface to map a function over a (A|C)ST
- A visitor is called on each node in some order
- A transformer visit each node and returns a node to take its place

The import system

A very complicated hammer for complex nails

The Import System

- Enable arbitrary rewriting of any file
- Can change everything from finding files to tokenization to parsing
- Requires a springboard import

```
import import_config
import dsl_module
```

- Implementation less crazy than it looks

References and Further Reading

- https://github.com/cdonovick/python-dsl-examples/
- https://docs.python.org/3/reference/datamodel.html
- https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/
- https://snarky.ca/tag/syntactic-sugar/
- https://greentreesnakes.readthedocs.io/
- https://www.youtube.com/watch?v=sPiWg5jSoZI

Tools

- astor: https://astor.readthedocs.io/en/latest/
- LibCST: https://libcst.readthedocs.io/en/latest/
- ast_tools: https://github.com/leonardt/ast_tools

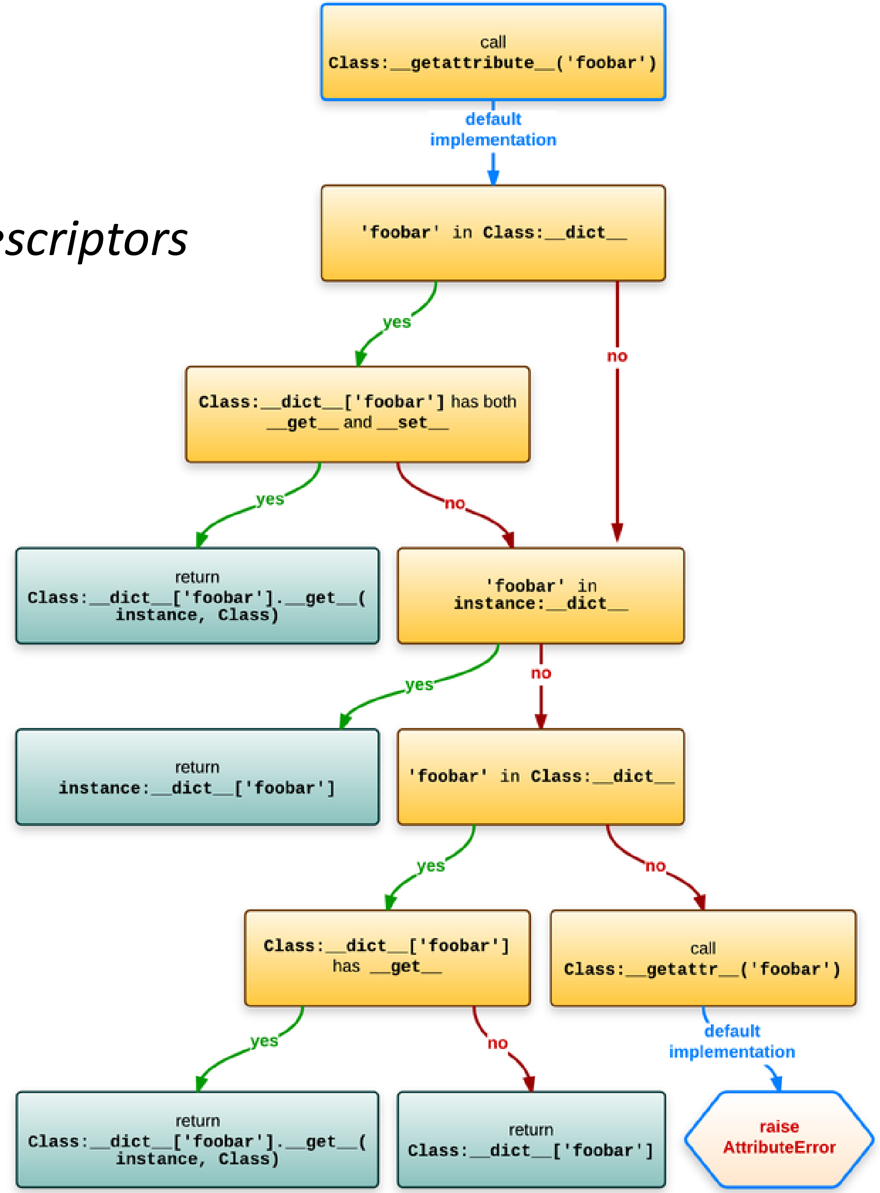
Understanding Attribute Access

```
bar = Foo()
--------------------    ------------------------------------------------------------
bar.x = 10              setattr(obj, attr, val):
                            #obj == bar, attr == 'x', val == 10
                            type(obj).__setattr__(obj, attr, val)


--------------------    ----------------------------------------------------------
bar.x                   getattr(obj, attr):
                            #obj == bar, attr == 'x'
                            return type(obj).__getattribute__(obj, attr)
```

The long road to attribute error

- Objects that define both `__get__` and `__set__` are *data descriptors*
- Objects that define just `__get__` are *non-data descriptors*
- Special methods are always looked up in the the class
- `__getattribute__` lookup order:
    - data descriptors
    - Instance `__dict__`
    - non-data descriptors
    - `__getattr__`
- `__setattr__` lookup order:
    - descriptors with `__set__` in class `__dict__`
    - Instance `__dict__`



https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/

What's so special about special methods?

- Operator overloading supported through `__special_methods__`
  - `__add__, __getitem__, __call__, ...`
- Look up of special method always goes through the class
  - ignores `class.__getattribute__` and `metaclass.__getattribute__`
  - Allows for optimization by storing special methods directly in the class struct
- "Work around" by modifying `obj.__class__`

modifying obj.__class__ ???

```python
class A:
    def __call__(self):
        print('calling an A')

def new_call(self):
    print('calling new_call')

a = A()
b = A()
c = A()

b.__call__ = new_call
c.__class__ = type('AWithNewCall', (A,), dict(__call__=new_call))

a() # calling an A
b() # calling an A
c() # calling new_call
```

The Way of the Descriptor

```
class cached_property(property):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.cached_values = weakref.WeakKeyDictionary()

    def __get__(self, obj, objtype=None):
        try: return self.cached_value[obj]
        except KeyError: pass
        val = super().__get__(obj, objtype)
        return self.cached_value.setdefault(obj, val)

    def __set__(self, obj, value):
        super().__set__(obj, value)
        self.cached_value.pop(obj, None)

    def __delete__(self, obj):
        super().__delete__(obj)
        self.cached_value.pop(obj, None)
```

Descriptors we all know and love

- `property`
- `__dict__`
- methods

```
class Int:
    def __init__(val):
        self.val = val

    def mul(self, other):
        return self.val * other

x = Int(2)
f = x.mul
l = list(map(f, [1,2,3]))
#l == [2,4,6]
```

Welcome to `__del__`

- Called when an object's reference count is 0 or when the garbage detector gets around to sweeping cycles
- NOT called on `del obj`, this just decrements the reference count
  - Cannot be used to break reference cycles
  - Use `weakref`
- Almost exclusively used for interacting with non-python objects
  - Although it may not be called on interpreter exit
- Few guarantees that any other object or module still exists when called
  - "Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the __del__() method is called."

C Bindings

- ctypes
  - Standard lib
  - Need to recreate structs in python
- cffi
  - On pypi
  - Fast
    - Can be compiled
  - Struct definitions can be loaded from header files
    - With many caveats
    - Fall back to writing C in strings - BAD