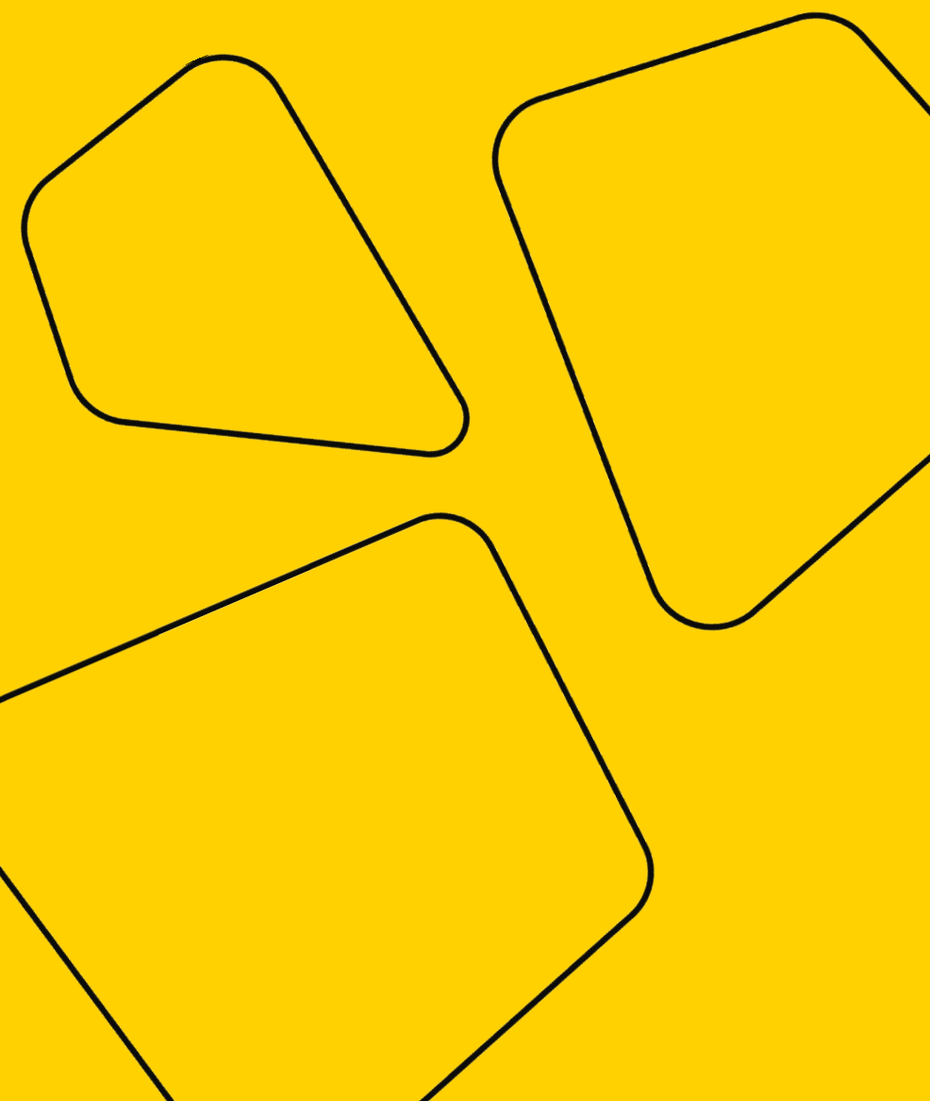
Classes

Software Development & Python

Nick Levashov, 2021



girafe
ai

Special attributes

girafe ai

User-defined functions

Attribute	Meaning	
`__doc__`	The function's documentation string, or `None` if unavailable; not inherited by subclasses.	Writable
`__name__`	The function's name.	Writable
`__qualname__`	The function's qualified name. *New in version 3.3.*	Writable
`__module__`	The name of the module the function was defined in, or `None` if unavailable.	Writable
`__defaults__`	A tuple containing default argument values for those arguments that have defaults, or `None` if no arguments have a default value.	Writable
`__code__`	The code object representing the compiled function body.	Writable

User-defined functions

`__globals__`	A reference to the dictionary that holds the function's global variables — the global namespace of the module in which the function was defined.	Read-only
`__dict__`	The namespace supporting arbitrary function attributes.	Writable
`__closure__`	`None` or a tuple of cells that contain bindings for the function's free variables. See below for information on the `cell_contents` attribute.	Read-only
`__annotations__`	A dict containing annotations of parameters. The keys of the dict are the parameter names, and `'return'` for the return annotation, if provided. For more information on working with this attribute, see Annotations Best Practices.	Writable
`__kwdefaults__`	A dict containing defaults for keyword-only parameters.	Writable

Custom classes

Special attributes:

`__name__`
: The class name.

`__module__`
: The name of the module in which the class was defined.

`__dict__`
: The dictionary containing the class's namespace.

`__bases__`
: A tuple containing the base classes, in the order of their occurrence in the base class list.

`__doc__`
: The class's documentation string, or `None` if undefined.

`__annotations__`
: A dictionary containing variable annotations collected during class body execution. For best practices on working with `__annotations__`, please see Annotations Best Practices.

Class instances

```
class A:
    def a(self):
        pass

instance = A()

instance.__class__
instance.__dict__

instance.a.__self__
```

Modules

Predefined (writable) attributes:

`__name__`
: The module's name.

`__doc__`
: The module's documentation string, or `None` if unavailable.

`__file__`
: The pathname of the file from which the module was loaded, if it was loaded from a file. The `__file__` attribute may be missing for certain types of modules, such as C modules that are statically linked into the interpreter. For extension modules loaded dynamically from a shared library, it's the pathname of the shared library file.

`__annotations__`
: A dictionary containing variable annotations collected during module body execution. For best practices on working with `__annotations__`, please see Annotations Best Practices.

Special read-only attribute: `__dict__` is the module's namespace as a dictionary object.

girafe ai

Special methods

Instance creation

```
object.__new__(cls[, ...])
object.__init__(self[, ...])
```

Instance deletion

```
object.__del__(self)
```

Instance deletion

```
class A:
    def __del__(self):
        print('deleted')


>>> a = T()
>>> del a
deleted
```

Instance deletion

```
class A:
    def __del__(self):
        print('deleted')
```

```
>>> a = T()
>>> del a
deleted

>>> a = T()
>>> b = a
>>> del a
>>> del b
deleted
```

Comparisons

girafe ai

Comparisons

```
object.__lt__(self, other)   # o < other
object.__le__(self, other)   # o <= other

object.__gt__(self, other)   # o > other
object.__ge__(self, other)   # o >= other

object.__eq__(self, other)   # o == other
object.__ne__(self, other)   # o != other
```

Comparisons

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @staticmethod
    def _validate_has_age(other):
        if not hasattr(other, 'age'):
            return NotImplemented

    def __eq__(self, other):
        self._validate_has_age(other)
        return self.age == other.age

    def __lt__(self, other):
        self._validate_has_age(other)
        return self.age < other.age
```

```
>>> p1 = Person("John", 36)
>>> p2 = Person("Mike", 24)
>>> print(p1 > p2)
True
```

Object representations

String representations

```
object.__repr__(self)    # repr(o)
                         # '<__main__.O object at 0x10bec2770>'

object.__str__(self)     # str(o)

object.__bytes__(self)   # bytes(o)

object.__format__(self, format_spec)  # format(o, format_spec)
```

Truth value testing

```
object.__bool__(self)  # bool(o)
```

Truth value testing

```
object.__bool__(self)   # bool(o)
```

```
object.__len__(self)    # len(o)
```

Hashing

```
object.__hash__(self)  # hash(o)
```

Hashing

```
object.__hash__(self)  # hash(o)
```

```
object.__eq__(self, other)
```

Numeric methods

girafe ai

Arithmetic operations

```
object.__add__(self, other)                    # o + other
object.__sub__(self, other)                    # o - other
object.__mul__(self, other)                    # o * other
object.__matmul__(self, other)                 # o @ other
object.__truediv__(self, other)                # o / other
object.__floordiv__(self, other)               # o // other
object.__mod__(self, other)                    # o % other
object.__divmod__(self, other)                 # divmod(o, other)
object.__pow__(self, other[, modulo])          # o ** other
                                               # pow(o, other)
                                               # pow(o, other, modulo)
object.__lshift__(self, other)                 # o << other
object.__rshift__(self, other)                 # o >> other
object.__and__(self, other)                    # o & other
object.__xor__(self, other)                    # o ^ other
object.__or__(self, other)                     # o | other
```

Reflected arithmetic operations

```
object.__radd__(self, other)                # other + o
object.__rsub__(self, other)                # other - o
object.__rmul__(self, other)                # other * o
object.__rmatmul__(self, other)             # other @ o
object.__rtruediv__(self, other)            # other / o
object.__rfloordiv__(self, other)           # other // o
object.__rmod__(self, other)                # other % o
object.__rdivmod__(self, other)             # divmod(other, o)
object.__rpow__(self, other[, modulo])      # other ** o
                                            # pow(other, o)
                                            # pow(other, o, modulo)

object.__rlshift__(self, other)             # other << o
object.__rrshift__(self, other)             # other >> o
object.__rand__(self, other)                # other & o
object.__rxor__(self, other)                # other ^ o
object.__ror__(self, other)                 # other | o
```

Reflected arithmetic operations

```
>>> from decimal import Decimal
>>> 1 + Decimal(10)
Decimal('11')
```

Reflected arithmetic operations

```
object.__radd__(self, other)                # other + o
object.__rsub__(self, other)                # other - o
object.__rmul__(self, other)                # other * o
object.__rmatmul__(self, other)             # other @ o
object.__rtruediv__(self, other)            # other / o
object.__rfloordiv__(self, other)           # other // o
object.__rmod__(self, other)                # other % o
object.__rdivmod__(self, other)             # divmod(other, o)
object.__rpow__(self, other[, modulo])      # other ** o
                                            # pow(other, o)
                                            # pow(other, o, modulo)

object.__rlshift__(self, other)             # other << o
object.__rrshift__(self, other)             # other >> o
object.__rand__(self, other)                # other & o
object.__rxor__(self, other)                # other ^ o
object.__ror__(self, other)                 # other | o
```

Augmented arithmetic assignments

```
object.__iadd__(self, other)                    # o += other
object.__isub__(self, other)                    # o -= other
object.__imul__(self, other)                    # o *= other
object.__imatmul__(self, other)                 # o @= other
object.__itruediv__(self, other)                # o /= other
object.__ifloordiv__(self, other)               # o //= other
object.__imod__(self, other)                    # o %= other
object.__ipow__(self, other[, modulo])          # o **= other
object.__ilshift__(self, other)                 # o <<= other
object.__irshift__(self, other)                 # o >>= other
object.__iand__(self, other)                    # o &= other
object.__ixor__(self, other)                    # o ^= other
object.__ior__(self, other)                     # o |= other
```

Augmented arithmetic assignments

```
>>> a = 1
>>> a += 1
```

Augmented arithmetic assignments

```
>>> a = 1
>>> a += 1

>>> t = (1, 2)
>>> t[0] += 1
```

Augmented arithmetic assignments

```
>>> a = 1
>>> a += 1

>>> t = (1, 2)
>>> t[0] += 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Augmented arithmetic assignments

```
>>> a = 1
>>> a += 1

>>> t = (1, 2)
>>> t[0] += 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> t = ([1], [2])
>>> t[0] += [3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Augmented arithmetic assignments

```
>>> a = 1
>>> a += 1

>>> t = (1, 2)
>>> t[0] += 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> t = ([1], [2])
>>> t[0] += [3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t
([1, 3], [2])
```

Augmented arithmetic assignments

```
object.__iadd__(self, other)                    # o += other
object.__isub__(self, other)                    # o -= other
object.__imul__(self, other)                    # o *= other
object.__imatmul__(self, other)                 # o @= other
object.__itruediv__(self, other)                # o /= other
object.__ifloordiv__(self, other)               # o //= other
object.__imod__(self, other)                    # o %= other
object.__ipow__(self, other[, modulo])          # o **= other
object.__ilshift__(self, other)                 # o <<= other
object.__irshift__(self, other)                 # o >>= other
object.__iand__(self, other)                    # o &= other
object.__ixor__(self, other)                    # o ^= other
object.__ior__(self, other)                     # o |= other
```

Unary arithmetic operations

```
object.__neg__(self)      # -o
object.__pos__(self)      # +o
object.__abs__(self)      # abs(o)
object.__invert__(self)   # ~o
```

Arithmetic types casting

```
object.__complex__(self)   # complex(o)
object.__int__(self)       # int(o)
object.__float__(self)     # float(o)
```

Arithmetic types casting

```
object.__complex__(self)   # complex(o)
object.__int__(self)       # int(o)
object.__float__(self)     # float(o)
```

```
object.__index__(self)
```

Arithmetic types casting

```
object.__complex__(self)   # complex(o)
object.__int__(self)       # int(o)
object.__float__(self)     # float(o)

object.__index__(self)

object.__round__(self[, ndigits])  # round(o)
                                   # round(o, ndigits)
object.__trunc__(self)             # math.trunc()
object.__floor__(self)             # math.floor()
object.__ceil__(self)              # math.ceil()
```

Attribute access

Attribute access

```
# default attribute access
object.__getattribute__(self, name)     # o.name

# fallback attribute access
object.__getattr__(self, name)          # o.name
```

```
object.__setattr__(self, name, value)  # o.name = value
object.__delattr__(self, name)          # del o.name

object.__dir__(self)                    # dir(o)
```

Descriptor

```
object.__get__(self, instance, owner=None)  # instance.o
object.__set__(self, instance, value)       # instance.o = value
object.__delete__(self, instance)           # del instance.o
```

Descriptor

```
class Meter:
    def __init__(self, value=0.0):
        self.value = float(value)
    def __get__(self, instance, owner=None):
        return self.value
    def __set__(self, instance, value):
        self.value = float(value)

class Foot:
    def __get__(self, instance, owner=None):
        return instance.meter * 3.2808
    def __set__(self, instance, value):
        instance.meter = float(value) / 3.2808

class Distance:
    meter = Meter()
    foot = Foot()
```

Descriptor

```
class Meter:
    def __init__(self, value=0.0):
        self.value = float(value)
    def __get__(self, instance, owner=None):
        return self.value
    def __set__(self, instance, value):
        self.value = float(value)

class Foot:
    def __get__(self, instance, owner=None):
        return instance.meter * 3.2808
    def __set__(self, instance, value):
        instance.meter = float(value) / 3.2808

class Distance:
    meter = Meter()
    foot = Foot()

>>> distance = Distance()
>>> distance.meter = 10
>>> print(distance.foot)
32.808
```

Descriptor

```
class Meter:
    def __init__(self, value=0.0):
        self.value = float(value)
    def __get__(self, instance, owner=None):
        return self.value
    def __set__(self, instance, value):
        self.value = float(value)

class Foot:
    def __get__(self, instance, owner=None):
        return instance.meter * 3.2808
    def __set__(self, instance, value):
        instance.meter = float(value) / 3.2808

class Distance:
    meter = Meter()
    foot = Foot()

>>> distance = Distance()
>>> distance.meter = 10
>>> print(distance.foot)
32.808
>>> distance.foot = 32.8
>>> print(distance.meter)
9.997561570348694
```

Descriptor protocol

Properties, bound methods, static methods, class methods are based on the descriptor protocol

```
object.__get__(self, instance, owner=None)   # instance.o
object.__set__(self, instance, value)        # instance.o = value
object.__delete__(self, instance)            # del instance.o
```

Descriptor protocol

Properties, bound methods, static methods, class methods are based on the descriptor protocol

```
object.__get__(self, instance, owner=None)   # instance.o
object.__set__(self, instance, value)        # instance.o = value
object.__delete__(self, instance)            # del instance.o
```

```
instance.method(*args, **kwargs)
```

Descriptor protocol

Properties, bound methods, static methods, class methods are based on the descriptor protocol

```
object.__get__(self, instance, owner=None)   # instance.o
object.__set__(self, instance, value)        # instance.o = value
object.__delete__(self, instance)            # del instance.o
```

```
instance.method(*args, **kwargs)

(instance.method)(*args, **kwargs)
```

Descriptor protocol

Properties, bound methods, static methods, class methods are based on the descriptor protocol

```
object.__get__(self, instance, owner=None)  # instance.o
object.__set__(self, instance, value)       # instance.o = value
object.__delete__(self, instance)           # del instance.o
```

```
instance.method(*args, **kwargs)

(instance.method)(*args, **kwargs)

method = instance.method
method(*args, **kwargs)
```

Callable objects

Callable objects

```
object.__call__(self[, args...])  # o()
```

Callable objects

```
def foo(): pass

foo()
foo.__call__()
```

Callable objects

```
def foo(): pass

foo()
foo.__call__()
```

```
class Foo:
    def __call__(self): pass

foo = Foo()
foo()
foo.__call__()
```

Container objects

girafe ai

Container objects

```
object.__len__(self)                          # len(0)

object.__getitem__(self, key)                 # o[key]
object.__setitem__(self, key, value)          # o[key] = value
object.__delitem__(self, key)                 # del o[key]
```

```
# o[key] when "o" is dict and key is not found by __getitem__
object.__missing__(self, key)

object.__iter__(self)                         # iter(o)
object.__reversed__(self)                     # reversed(o)
object.__contains__(self, item)               # item in o
```

Containers and iterators

```
# Iterable
container.__iter__(self)   # iter(c)

# Iterator protocol
iterator.__iter__(self)    # iter(i)
iterator.__next__(self)    # next(i)
```

Reference

- https://docs.python.org/3/reference/datamodel.html
- https://rszalski.github.io/magicmethods/
- https://docs.python.org/3/howto/descriptor.html

Class decorators

girafe ai

Decorator as a Class

Decorator as a Class

```
def log_decorator(log=True):

    def helper_decorator(func):

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            if log:
                print(f'Called with {args} {kwargs}')
            return func(*args, **kwargs)

        return wrapper

    return helper_decorator


@log_decorator(log=True)
def test(*args, **kwargs):
    pass

>>> test(1, b=2)
Called with (1,) {'b': 2}
```

Decorator as a Class

```
class LogDecorator:
    def __init__(self, log=True):
        self.log = log

    def __call__(self, func):

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            if self.log:
                print(f'Called with {args} {kwargs}')
            return func(*args, **kwargs)

        return wrapper


@LogDecorator(log=True)
def test(*args, **kwargs):
    pass

>>> test(1, b=2)
Called with (1,) {'b': 2}
```

Decorating a Class

Decorating a Class

```
def add_print_method(cls): pass  # TODO
```

```
@add_print_method
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age


>>> p = Person('John', 36)
>>> p.print()
Person: {'name': 'John', 'age': 36}
```

Decorating a Class

```
def add_print_method(cls):
    def print_method(self):
        print(f'{self.__class__.__name__}: {self.__dict__}')
    cls.print = print_method
    return cls


@add_print_method
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age


>>> p = Person('John', 36)
>>> p.print()
Person: {'name': 'John', 'age': 36}
```

```
from functools import total_ordering

@total_ordering
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @staticmethod
    def _validate_has_age(other):
        if not hasattr(other, 'age'):
            return NotImplemented

    def __eq__(self, other):
        self._validate_has_age(other)
        return self.age == other.age

    def __lt__(self, other):
        self._validate_has_age(other)
        return self.age < other.age


>>> p1 = Person("John", 36)
>>> p2 = Person("Mike", 24)
>>> print(p1 >= p2)
True
```

Dataclasses

```python
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

Dataclasses

```python
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def __post_init__(self):
        if self.quantity_on_hand < 0:
            raise ValueError('quantity_on_hand must be >= 0')

def total_cost(self) -> float:
    return self.unit_price * self.quantity_on_hand
```

Dataclasses

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def __post_init__(self):
        if self.quantity_on_hand < 0:
            raise ValueError('quantity_on_hand must be >= 0')

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

```
>>> item1 = InventoryItem('Phone', 1699)
>>> item1.quantity_on_hand += 1
>>> item2 = InventoryItem('Phone case', 59.99, 2)
>>> print(item2.total_cost())
119.98
```

girafe
ai