2023 12December 18

December 21, 2023

1 Agenda

1. Inheritance
2. Magic methods (`__del__`)
3. Object system
4. Metaclasses
5. Iterators etc.

[2]:
```
class Person:

    def __init__(self, name):
        self.name = name

    def greet(self):
        return f'Hello, {self.name}!'

p1 = Person('name1')
p2 = Person('name2')

print(p1.greet())
print(p2.greet())

class Employee(Person):     # Employee is-a Person, i.e., inherits from Person

    def __init__(self, name, id_number):
        # Person.__init__(self, name)
        super().__init__(name)          # do what my parent does in __init__...
        self.id_number = id_number      # add my own things

e1 = Employee('emp1', 1)# e1 has __init__? no. Employee has __init__? Yes
e2 = Employee('emp2', 2)

print(e1.greet()) # e1 has greet? No. Employee has greet? No. Person has greet?␣
 ↪yes
print(e2.greet())
```

```
Hello, name1!
```

```
Hello, name2!
Hello, emp1!
Hello, emp2!
```

2 Inheritance

All inheritance is based on the search for attributes. When we look for an attribute in a Python object (an instance, that is), Python searches in the following order:

- i – the instance itself
- c – the class of the instance (`type(i)`)
- p – the parent of the class
- o – `object`, the top object in the class

This means, in practice:

- If we have the same method in both a child class and a parent class, then we can remove the child class implementation, and rely on the parent class
- If we write a method in the child class, then that takes priority, and the parent class's method is never run.
- If we want to combine the method in the child class with the parent class, then we have a few options:
  1. Copy the code from the parent class into the child class. There are a number of problems with doing it this way – not recommended.
  2. Call the parent method explicitly (`Class.method(self, arg1)`). This way, the parent class gets to run first, and then we add functionality in the child class.
  3. The most modern way is to use `super`, as in `super().method(arg1)`. We don't need to pass `self` here! Once again, we normally do this at first in the method, and then have more specific instructions in our method.

[3]:
```python
class First:
    def __init__(self, x):
        self.x = x

    def x2(self):
        return self.x * 2

class Second:
    def __init__(self, y):
        self.y = y

    def y3(self):
        return self.y * 3

class Third(First, Second):
    pass
```

[4]:
```python
# who does Person inherit from?  We can always check __bases__
Person.__bases__
```

[4]: (object,)

[5]:
```python
# What about Person's MRO (method resolution order)
Person.__mro__
```

[5]: (__main__.Person, object)

[6]:
```python
Employee.__bases__
```

[6]: (__main__.Person,)

[7]:
```python
Employee.__mro__
```

[7]: (__main__.Employee, __main__.Person, object)

[8]:
```python
First.__bases__
```

[8]: (object,)

[9]:
```python
First.__mro__
```

[9]: (__main__.First, object)

[10]:
```python
Second.__bases__
```

[10]: (object,)

[11]:
```python
Second.__mro__
```

[11]: (__main__.Second, object)

[12]:
```python
Third.__bases__
```

[12]: (__main__.First, __main__.Second)

[13]:
```python
Third.__mro__
```

[13]: (__main__.Third, __main__.First, __main__.Second, object)

[14]:
```python
# what happens when we create an instance of Third?

t = Third()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[14], line 3
      1 # what happens when we create an instance of Third?
----> 3 t = Third()
```

```
TypeError: First.__init__() missing 1 required positional argument: 'x'
```

[15]:
```
t = Third(10)
```

[16]:
```
vars(t)
```

[16]:
```
{'x': 10}
```

[17]:
```
t.x2()   # t has x2? No. Third has x2? No. First has x2? Yes...
```

[17]:
```
20
```

[18]:
```
t.y3()   # t has y3? No. Third has y3? No. First has y3? No. Second has y3? Yes
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[18], line 1
----> 1 t.y3()   # t has y3? No. Third has y3? No. First has y3? No. Second has␣
 ↪y3? Yes

Cell In[3], line 13, in Second.y3(self)
     12 def y3(self):
---> 13     return self.y * 3

AttributeError: 'Third' object has no attribute 'y'
```

[20]:
```
class BadClass(First, Third, Second):
    pass
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[20], line 1
----> 1 class BadClass(First, Third, Second):
      2     pass

TypeError: Cannot create a consistent method resolution
order (MRO) for bases First, Third, Second
```

[ ]:
```
class First:
    def __init__(self, x):
        self.x = x

    def x2(self):
        return self.x * 2
```

```python
class Second:
    def __init__(self, y):
        self.y = y

    def y3(self):
        return self.y * 3

class Third(First, Second):
    def __init__(self, x, y):
        super().__init__(self, x)
        # First.__init__(self, x)
        # Second.__init__(self, y)
```

[21]:
```python
help(super)
```

```
Help on class super in module builtins:

class super(object)
 |  super() -> same as super(__class__, <first argument>)
 |  super(type) -> unbound super object
 |  super(type, obj) -> bound super object; requires isinstance(obj, type)
 |  super(type, type2) -> bound super object; requires issubclass(type2, type)
 |  Typical use to call a cooperative superclass method:
 |  class C(B):
 |      def meth(self, arg):
 |          super().meth(arg)
 |  This works for class methods too:
 |  class C(B):
 |      @classmethod
 |      def cmeth(cls, arg):
 |          super().cmeth(arg)
 |
 |  Methods defined here:
 |
 |  __get__(self, instance, owner=None, /)
 |      Return an attribute of instance, which is of type owner.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
```

```
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __self__
 |      the instance invoking super(); may be None
 |
 |  __self_class__
 |      the type of the instance invoking super(); may be None
 |
 |  __thisclass__
 |      the class invoking super()
```

[32]:

```python
class First:
    def __init__(self, x):
        self.x = x

    def x2(self):
        return self.x * 2

class Second(First):
    def __init__(self, y):
        super().__init__(y)
        self.y = y

    def y3(self):
        return self.y * 3

class Third(Second):
    def __init__(self, x):
        super().__init__(x)
```

[31]:

```python
t = Third(17)
```

[33]:

```python
vars(t)
```

[33]:

```
{'x': 17, 'y': 17}
```

[34]:

```python
object
```

[34]:

```
object
```

[38]:
```python
str(t) # __str__() on t? No. __str__ on Third? No. __str__ on Second? no.␣
↪__str__ on First? No.
       # ___str__ on object? Yes!
```

[38]: '<__main__.Third object at 0x11237db10>'

[36]:
```python
0x11237db10
```

[36]: 4600617744

[37]:
```python
id(t)
```

[37]: 4600617744

[39]:
```python
object.__str__(t)
```

[39]: '<__main__.Third object at 0x11237db10>'

[40]:
```python
t.x2()    # --> Third.x2(t)
```

[40]: 34

[42]:
```python
class First:
    def __init__(self, x):
        self.x = x

    def x2(self):
        return self.x * 2

class Second(First):
    def __init__(self, y):
        super().__init__(y)
        self.y = y

    def y3(self):
        return self.y * 3

class Third(Second):
    def __init__(self, x):
        super().__init__(x)
    def __str__(self):
        return f'My Third has attributes: {vars(self)}'
```

[43]:
```python
def myfunc():
    asdfsadfsafffsa
```

[44]:
```python
def myfunc():
    asdfsadfsafffsa
```

```
    asdfakjfhskjfshjkfha
```

[46]:
```
def myfunc():
    asdfsadfsafffsa
  asdfakjfhskjfshjkfha
```

```
  File <tokenize>:3
    asdfakjfhskjfshjkfha
    ^
IndentationError: unindent does not match any outer indentation level
```

[47]:
```
t = Third(123)
```

[48]:
```
print(t)  # when I run print, it really runs print(str(ARG))
```

```
My Third has attributes: {'x': 123, 'y': 123}
```

3 Exercise: Big Bowl

A big bowl is just like a bowl, but takes up to 5 scoops, not just 3. Define `BigBowl` while making as few changes to `Bowl` as possible, and also writing as little code as possible.

[78]:
```
class Scoop:
    def __init__(self, flavor):
        self.flavor = flavor

    def __repr__(self):
        return f'Scoop of {self.flavor}'

class Bowl:
    MAX_SCOOPS = 3

    def __init__(self):
        self.scoops = []

    def add_scoops(self, *new_scoops):
        for one_scoop in new_scoops:
            if len(self.scoops) < self.MAX_SCOOPS:
                self.scoops.append(one_scoop)

    def flavors(self):
        return [one_scoop.flavor
                for one_scoop in self.scoops]

    def __repr__(self):
```

```
        output = 'Bowl of: \n'

        # for index, one_scoop in enumerate(self.scoops, 1):
        #     output += f'\t{index}: {one_scoop}\n'

        # return output

        return output + '\n'.join([f'\t{index}: {one_scoop}'
                                   for index, one_scoop in enumerate(self.
↪scoops, 1)])

class BigBowl(Bowl):
    MAX_SCOOPS = 5

s1 = Scoop('chocolate')
s2 = Scoop('vanilla')
s3 = Scoop('coffee')
s4 = Scoop('flavor 4')
s5 = Scoop('flavor 5')

b = Bowl()
b.add_scoops(s1, s2)
b.add_scoops(s3)
b.add_scoops(s4, s5)
print(b.flavors())

bb = BigBowl()
bb.add_scoops(s1, s2)
bb.add_scoops(s3)
bb.add_scoops(s4, s5)
print(bb.flavors())
```

```
['chocolate', 'vanilla', 'coffee']
['chocolate', 'vanilla', 'coffee', 'flavor 4', 'flavor 5']
```

4 Exercise: Printing our ice cream

1. Implement `__str__` on `Scoop` such that printing / calling `str` on an instance of `Scoop` returns a string like "Scoop of chocolate".
2. Implement `__str__` on `Bowl` such that printing / calling returns a string like:

Bowl of: 1. Scoop of chocolate 2. Scoop of vanilla 3. Scoop of coffee

[79]:
```
print(s1) # print(str(s1)) -> print(s1.__str__()) -> does s1 have __str__? No.␣
↪Does Scoop have __str__? No.
          #         does object have __str__? Yes!
```

```
Scoop of chocolate
```

[80]:
```
print(b)
```

```
Bowl of:
	1: Scoop of chocolate
	2: Scoop of vanilla
	3: Scoop of coffee
```

[81]:
```
s1
```

[81]:
```
Scoop of chocolate
```

[82]:
```
b
```

[82]:
```
Bowl of:
	1: Scoop of chocolate
	2: Scoop of vanilla
	3: Scoop of coffee
```

5 `__str__` and `__repr__`

These two methods both get `self` as an argument, and are both supposed to return a string.

`__str__` is meant for end users, `__repr__` is meant for behind-the-scenes debugging and printing.

- If I don't define `__repr__`, then I get the default.
- If I don't define `__str__`, but `__repr__` is defined, then it is used.

I suggest always defining `__repr__`, and only defining `__str__` if and when you need.

[ ]:
```

```

[83]:
```
object.__str__(s1)
```

[83]:
```
'Scoop of chocolate'
```

[84]:
```
object.__repr__(s1)
```

[84]:
```
'<__main__.Scoop object at 0x112374910>'
```

6 Next up

1. Magic methods + overloading
2. `__del__`
3. The Python object system
4. Metaclasses
5. Iteration

Resume at :35

[86]:
```python
class MyClass:
    def __init__(self, x):
        self.x = x

    def __len__(self):
        return len(self.x)

m1 = MyClass('abcde')

len(m1)  # this actually calls m1.__len__()  -> len('abcde') --> 'abcde'.
 ↪__len__()
```

[86]: 5

[87]:
```python
m2 = MyClass(100)
len(m2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[87], line 2
      1 m2 = MyClass(100)
----> 2 len(m2)

Cell In[86], line 6, in MyClass.__len__(self)
      5 def __len__(self):
----> 6     return len(self.x)

TypeError: object of type 'int' has no len()
```

[88]:
```python
'abcde'.__len__()
```

[88]: 5

[89]:
```python
m2.x.__len__()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[89], line 1
----> 1 m2.x.__len__()

AttributeError: 'int' object has no attribute '__len__'
```

[90]:
```python
m1 = MyClass('abcde')
m2 = MyClass('abcde')
```

```
m1 == m2
```

[90]: False

[91]:
```
# when we run ==, Python runs the __eq__ method
# it runs the method on the left-side argument

m1.__eq__(m2)
```

[91]: NotImplemented

[92]:
```
type(NotImplemented)
```

[92]: NotImplementedType

[105]:
```
from functools import total_ordering

@total_ordering
class MyClass:
    def __init__(self, x):
        self.x = x

    def __len__(self):
        return len(self.x)

    def __eq__(self, other):
        # return vars(self) == vars(other)
        if hasattr(other, 'x'):
            return self.x == other.x
        return False

    def __lt__(self, other):
        if hasattr(other, 'x'):
            return self.x < other.x
        return False

m1 = MyClass('abcde')
m2 = MyClass('abcde')

m1 == m2
```

[105]: True

[106]:
```
m1 == 100
```

[106]: False

[107]:
```
100 == m1
```

[107]: False

[108]:
```
m3 = MyClass('bcdef')

m1 < m3
```

[108]: True

[109]:
```
m1 >= m3
```

[109]: False

[110]:
```
# let's try addition!

m1 + m2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[110], line 3
      1 # let's try addition!
----> 3 m1 + m2

TypeError: unsupported operand type(s) for +: 'MyClass' and 'MyClass'
```

[111]:
```
10 + '20'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[111], line 1
----> 1 10 + '20'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

[112]:
```
'10' + 20
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[112], line 1
----> 1 '10' + 20

TypeError: can only concatenate str (not "int") to str
```

[135]:
```
from functools import total_ordering
```

```python
@total_ordering
class MyClass:
    def __init__(self, x):
        self.x = x

    def __len__(self):
        return len(self.x)

    def __eq__(self, other):
        # return vars(self) == vars(other)
        if hasattr(other, 'x'):
            return self.x == other.x
        return False

    def __lt__(self, other):
        if hasattr(other, 'x'):
            return self.x < other.x
        return False

    def __add__(self, other):        # self + other
        if hasattr(other, 'x'):      # does other even have an 'x' attribute?
            return MyClass(self.x + other.x)
        return MyClass(self.x + str(other))

    def __radd__(self, other):   # reverse add!
        if hasattr(other, 'x'):
            return MyClass(other.x + self.x)
        return MyClass(str(other) + self.x)

    def __iadd__(self, other):   # in-place add, +=
        if hasattr(other, 'x'):
            self.x += other.x
        self.x = self.x + str(other)
        return self

    def __repr__(self):
        return f'Person with {self.x=}'

m1 = MyClass('abcde')
m2 = MyClass('fghij')

m1 + m2
```

[135]: Person with self.x='abcdefghij'

[136]:
```python
m1 + 100
```

[136]: Person with self.x='abcde100'

Other methods for operators:

- + – `__add__`
- - – `__sub__`
- * – `__mul__`
- / – `__truediv__`
- // – `__floordiv__`
- ** – `__exp__`
- % – `__mod__`

[137]:
```
[10, 20, 30] + [40, 50, 60]
```

[137]: [10, 20, 30, 40, 50, 60]

[138]:
```
m1 + 100
```

[138]: Person with self.x='abcde100'

[139]:
```
100 + m1
```

[139]: Person with self.x='100abcde'

[140]:
```
# what's going to happen here?

print(id(m1))
m1 += 'xyz'    # m1 = m1 + 'xyz'
print(id(m1))
```

```
4609949072
4609949072
```

[141]:
```
print(m1)
```

```
Person with self.x='abcdexyz'
```

[ ]:
```

```

[142]:
```
dir(5)
```

[142]:
```
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__divmod__',
```

```
'__doc__',
'__eq__',
'__float__',
'__floor__',
'__floordiv__',
'__format__',
'__ge__',
'__getattribute__',
'__getnewargs__',
'__getstate__',
'__gt__',
'__hash__',
'__index__',
'__init__',
'__init_subclass__',
'__int__',
'__invert__',
'__le__',
'__lshift__',
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__neg__',
'__new__',
'__or__',
'__pos__',
'__pow__',
'__radd__',
'__rand__',
'__rdivmod__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rfloordiv__',
'__rlshift__',
'__rmod__',
'__rmul__',
'__ror__',
'__round__',
'__rpow__',
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
```

```
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__truediv__',
 '__trunc__',
 '__xor__',
 'as_integer_ratio',
 'bit_count',
 'bit_length',
 'conjugate',
 'denominator',
 'from_bytes',
 'imag',
 'numerator',
 'real',
 'to_bytes']
```

[144]:
```
x = 123
y = [10, 20, 30]

# if I want to debug
print(f'x = {x}, y = {y}')
```

```
x = 123, y = [10, 20, 30]
```

[145]:
```
# as of Python 3.10
print(f'{x=}, {y=}')
```

```
x=123, y=[10, 20, 30]
```

[146]:
```
print(f'{len(y)=}')
```

```
len(y)=3
```

[147]:
```
# have you noticed that all of these types use []
s = 'abcde'
s[3]
```

[147]: 'd'

[148]:
```
mylist = [10, 20, 30, 40, 50]
mylist[3]
```

[148]: 40

[149]:
```
d = {'a':10, 'b':20, 'c':30}
d['b']
```

[149]: 20

[150]:
```python
# all of these implement __getitem__, that take self and the index

class MyClass:
    def __init__(self, x):
        self.x = x

    def __repr__(self):
        return f'Person with {self.x=}'

    def __getitem__(self, index):
        return self.x[index]

m = MyClass('abcde')
m[3]
```

[150]: 'd'

7 Exercise: Magic methods and bowls

1. Make it possible to use `len` on an instance of `Bowl`, getting the number of scoops.
2. Make it possible to use `[]` on a `Bowl` instance, getting back one scoops.
3. Make it possible to use `+` on two `Bowl` instances, getting back one with the scoops from both.

[170]:
```python
class Scoop:
    def __init__(self, flavor):
        self.flavor = flavor

    def __repr__(self):
        return f'Scoop of {self.flavor}'

class Bowl:
    MAX_SCOOPS = 3

    def __init__(self):
        self.scoops = []

    def add_scoops(self, *new_scoops):
        for one_scoop in new_scoops:
            if len(self.scoops) < self.MAX_SCOOPS:
                self.scoops.append(one_scoop)

    def flavors(self):
        return [one_scoop.flavor
                for one_scoop in self.scoops]

```

```
    def __repr__(self):
        output = 'Bowl of: \n'

        return output + '\n'.join([f'\t{index}: {one_scoop}'
                                   for index, one_scoop in enumerate(self.
↪scoops, 1)])

    def __len__(self):
        return len(self.scoops)

    def __getitem__(self, index):
        if isinstance(index, slice):
            b = Bowl()
            b.scoops = self.scoops[index]    # index is a slice, get that part␣
↪of self.scoops
            return b
        return self.scoops[index]    # index is an integer (we hope), and return␣
↪that scoop

    def __add__(self, other):
        if not isinstance(other, Bowl):
            raise TypeError('Can only add bowls to other bowls')

        b = Bowl()
        b.add_scoops(*(self.scoops + other.scoops))
        return b

s1 = Scoop('chocolate')
s2 = Scoop('vanilla')
s3 = Scoop('coffee')
s4 = Scoop('flavor 4')
s5 = Scoop('flavor 5')

b = Bowl()
b.add_scoops(s1, s2)
b.add_scoops(s3)
b.add_scoops(s4, s5)
print(b.flavors())

print(len(b))
b[1]
```

```
['chocolate', 'vanilla', 'coffee']
3
```

[170]:
```
Scoop of vanilla
```

[171]:
```python
b[:2]
```

[171]:
```
Bowl of:
        1: Scoop of chocolate
        2: Scoop of vanilla
```

[172]:
```python
b1 = Bowl()
b1.add_scoops(s1)

b2 = Bowl()
b2.add_scoops(s2, s3, s4)

b1 + b2
```

[172]:
```
Bowl of:
        1: Scoop of chocolate
        2: Scoop of vanilla
        3: Scoop of coffee
```

[173]:
```python
# __del__

# this method runs when the reference count drops to zero

x = [10, 20, 30]    # refcount to [10, 20, 30] is 1
y = x      # refcount is 2
z = x      # refcount is 3
```

[174]:
```python
x = None
y = None
z = None

# what is the refcount of [10, 20, 30]?  0
# when it goes to 0, the object is deleted and the memory is freed
```

[175]:
```python
class MyClass:
    def __init__(self, x):
        self.x = x

    def __del__(self):
        print(f'__del__ ran; {self.x=}')

m1 = MyClass(10)
m2 = MyClass(20)
m3 = MyClass(30)
```

[176]:
```python
m1 = MyClass(40)
```

```
__del__ ran; self.x=10
```

8 Who does use `__del__`?

1. Files – when they are freed, they are flushed and closed
2. NumPy arrays – when their refcount goes to 0, they free the C part of the memory usage
3. TempFile – erases the tempfile when the refcount goes to 0

[177]:
```
import gc
```

[180]:
```
x = [10, 20, 30, 40, 50]
len(gc.get_referrers(x))
```

[180]: 1

[181]:
```
y = x
```

[182]:
```
len(gc.get_referrers(x))
```

[182]: 1

[183]:
```
help(gc.get_referrers)
```

```
Help on built-in function get_referrers in module gc:

get_referrers(…)
    get_referrers(*objs) -> list
    Return the list of objects that directly refer to any of objs.
```

[184]:
```
y = [x]
```

[185]:
```
len(gc.get_referrers(x))
```

[185]: 2

9 Next time

1. Object system in Python
2. Metaclasses
3. Iterators

[186]:
```
gc.collect()
```

[186]: 2701

[187]:
```
x = [10, 20, 30, 40, 50]

del(x)    # this removes the variable x
```

[ ]:
```
```