Inheritance

November 29, 2019

0.1 Inheritance

Other languages automatically call base class initializers. However, Python treats `__init__()` like any other method.

Base class `__init__()` is not called if overridden. Use `super()` to call base class `__init__()`.

[1]:
```
class SimpleList:
    def __init__(self, items):
        self._items = list(items)

    def add(self, item):
        self._items.append(item)

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

    def sort(self):
        self._items.sort()

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

    def __repr__(self):
        return 'SimpleList({!r})'.format(self._items)
```

[2]:
```
class SortedList(SimpleList):
    def __init__(self, items=()):
        super().__init__(items)
        self.sort()

    def add(self, item):
        super().add(item)
        self.sort()

    def __repr__(self):
        return 'SortedList({!r})'.format(list(self))
```

[3]:
```
sl = SortedList([1, 3, 6, 2, 4, 7, 5, 9])
print(sl)
sl.add(-1)
sl.add(10)
print(sl)
```

```
SortedList([1, 2, 3, 4, 5, 6, 7, 9])
SortedList([-1, 1, 2, 3, 4, 5, 6, 7, 9, 10])
```

0.2 isinstance()

Determines if an object is of a specified type. `isinstance()` will return `True` if the object is an instance of a subclass of the second argument.

In addition, a tuple of types can be passed as the second argument. This is equivalent to asking if the first argument is an instance of any of the types in the tuple.

[4]:
```
isinstance(3, int)
```

[4]: True

[5]:
```
isinstance('hello', str)
```

[5]: True

[6]:
```
isinstance(4.567, bytes)
```

[6]: False

[7]:
```
isinstance(sl, SortedList)
```

[7]: True

[8]:
```
isinstance(sl, SimpleList)
```

[8]: True

[9]:
```
x = []
isinstance(x, (float, dict, list))
```

[9]: True

[10]:
```
class IntList(SimpleList):
    def __init__(self, items=()):
        for x in items: self._validate(x)
        super().__init__(items)

    @staticmethod
```

```
    def _validate(x):
        if not isinstance(x, int):
            raise TypeError('IntList only accepts int values')

    def add(self, item):
        self._validate(item)
        super().add(item)

    def __repr__(self):
        return 'IntList({!r})'.format(list(self))
```

[11]:
```
il = IntList([1, 3, 2, 4])
il.add(19)
print(il)
```

```
IntList([1, 3, 2, 4, 19])
```

0.3 issubclass()

Determines if a type is a subclass of another.

[12]:
```
issubclass(IntList, SimpleList)
```

[12]: True

[13]:
```
issubclass(SortedList, SimpleList)
```

[13]: True

[14]:
```
issubclass(SortedList, IntList)
```

[14]: False

[15]:
```
class MyInt(int): pass
class MyVerySpecialInt(MyInt): pass
```

[16]:
```
issubclass(MyVerySpecialInt, int)
```

[16]: True

0.4 Multiple Inheritance

Defining a class with more than one base class.

If a class has multiple base classes and doesn't define its `__init__()` method. Then, **only the initializer of the first base class is automatically called**.

The list of base classes of a type is stored in the `__bases__` member.

[17]:
```
class SortedIntList(IntList, SortedList):
    def __repr__(self):
        return 'SortedIntList({!r})'.format(list(self))
```

[18]:
```
sil = SortedIntList([1, 3, 2, 4, 6, 5])
print(sil)
sil.add(-1234)
print(sil)
```

```
SortedIntList([1, 2, 3, 4, 5, 6])
SortedIntList([-1234, 1, 2, 3, 4, 5, 6])
```

[19]:
```
SortedIntList([1, 3, 2, '4', 6])
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-19-59656f4f0ffa> in <module>
----> 1 SortedIntList([1, 3, 2, '4', 6])


<ipython-input-10-b7ed9fffcde6> in __init__(self, items)
  1 class IntList(SimpleList):
  2     def __init__(self, items=()):
----> 3         for x in items: self._validate(x)
  4         super().__init__(items)
  5


<ipython-input-10-b7ed9fffcde6> in _validate(x)
  7     def _validate(x):
  8         if not isinstance(x, int):
----> 9             raise TypeError('IntList only accepts int values')
 10
 11     def add(self, item):


TypeError: IntList only accepts int values
```

[20]:
```
class Base1:
    def __init__(self):
        print('Base1.__init__')
```

```
class Base2:
    def __init__(self):
        print('Base2.__init__')

class Sub(Base1, Base2):
    pass
```

[21]:
```
s = Sub()
```

```
Base1.__init__
```

[22]:
```
Sub.__bases__
```

[22]:
```
(__main__.Base1, __main__.Base2)
```

0.5 Method Resolution Order (MRO)

The **MRO** of a class is the ordering of a class' inheritance graph used to determine **which implementation to use when a method is invoked**.

When you invoke a method on an object which has one or more base classes, the actual code that gets run may be defined in multiple places.

The **MRO** is the way Python resolves the **diamond problem of inheritance**

The **MRO** of a type is stored in the `__mro__` member or calling the `mro()` method.

0.5.1 C3

Python uses the **C3** algorithm to build the **MRO** in a deterministic way.

- Subclasses come **before** base classes
- Base class order from class definition is **preserved**
- First two qualities are preserved **no matter** where you start in the inheritance graph

[23]:
```
SortedList.__mro__
```

[23]:
```
(__main__.SortedList, __main__.SimpleList, object)
```

[24]:
```
class A:
    def func(self):
        return 'A.func'

class B(A):
    def func(self):
        return 'B.func'

class C(A):
    def func(self):
```

```
        return 'C.func'

class D(B, C):
    pass
```

[25]:
```
D.__mro__
```

[25]:
```
(__main__.D, __main__.B, __main__.C, __main__.A, object)
```

[26]:
```
D().func()
```

[26]:
```
'B.func'
```

[27]:
```
class E(C, B):
    pass
```

[28]:
```
E.__mro__
```

[28]:
```
(__main__.E, __main__.C, __main__.B, __main__.A, object)
```

[29]:
```
E().func()
```

[29]:
```
'C.func'
```

[30]:
```
class AA: pass
class BB(AA): pass
class CC(AA): pass
class DD(BB, AA, CC): pass
```

```
      ␣
↪---------------------------------------------------------------------------

      TypeError                                 Traceback (most recent call␣
↪last)

      <ipython-input-30-c09699461d8f> in <module>
        2 class BB(AA): pass
        3 class CC(AA): pass
   ----> 4 class DD(BB, AA, CC): pass


      TypeError: Cannot create a consistent method resolution
   order (MRO) for bases AA, CC
```

0.6 `super()`

Given an MRO and a class C, `super()` gives you an object which resolves methods using only the part of the MRO which comes after C.

`super()` returns a **proxy** object which **routes** method calls.

- **Bound proxy** is bound to an specific class or instance.
- **Unbound proxy** is not bound to a class or instance.

0.6.1 Bound Proxy

There are two types of bound proxies **instance-bound** and **class-bound**.

Class-bound proxy To create a **class-bound** proxy the both parameters of `super()` must be class objects. The second class argument must be a subclas of or the same class as the first argument.

```
super(base-class, derived-class)
```

Python finds the **MRO** of the `derived-class`, afterwards it finds `base-class` in that **MRO**. Then, it takes everything **after** `base-class` in the **MRO**, and finds the first class in that sequence with matching method signature.

Instance-bound proxy **Instance-bound** proxy behave similarly to the **class-bound** proxy, except they bind to an instance.

To create an **instance-bound** proxy the second argument of `super()` must be an instance of a class of the same class as the first argument or any class derived from it.

Python finds the type of the second argument and calculates its **MRO**, afterwards it finds the location of the first argument in that **MRO**. Then, uses everything **after** that location to find the matching method signature.

0.6.2 Unbound proxy

TBD

0.6.3 `super()` without parameters

When `super()` is called without parameters Python sorts out the arguments on its own.

When called from inside an **instance method** Python does the following call: `super(class-of-method, self)`. Therefore, it's using an **instance-bound** proxy.

When called from inside a **class method** Python does the following call: `super(class-of-method, class)`. Therefore, it's using a **class-bound** proxy.

[31]:
```
SortedIntList.mro()
```

[31]:
```
[__main__.SortedIntList,
 __main__.IntList,
 __main__.SortedList,
```

```
 __main__.SimpleList,
 object]
```

[32]:
```
super(SortedList, SortedIntList)
```

[32]:
```
<super: __main__.SortedList, __main__.SortedIntList>
```

[33]:
```
super(SortedList, SortedIntList).add
```

[33]:
```
<function __main__.SimpleList.add(self, item)>
```

[34]:
```
super(SortedList, SortedIntList).add(4)
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-34-565a4aa6b79a> in <module>
   ----> 1 super(SortedList, SortedIntList).add(4)


       TypeError: add() missing 1 required positional argument: 'item'
```

[35]:
```
super(SortedIntList, SortedIntList)._validate(5)
super(SortedIntList, SortedIntList)._validate('5')
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-35-fdaed98f0411> in <module>
         1 super(SortedIntList, SortedIntList)._validate(5)
   ----> 2 super(SortedIntList, SortedIntList)._validate('5')


       <ipython-input-10-b7ed9fffcde6> in _validate(x)
         7     def _validate(x):
         8         if not isinstance(x, int):
   ----> 9             raise TypeError('IntList only accepts int values')
        10
```

```
     11     def add(self, item):

TypeError: IntList only accepts int values
```

[36]:
```
super(int, IntList)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-36-445c7c396676> in <module>
----> 1 super(int, IntList)

TypeError: super(type, obj): obj must be an instance or subtype of type
```

[37]:
```
SortedIntList.mro()
```

[37]:
```
[__main__.SortedIntList,
 __main__.IntList,
 __main__.SortedList,
 __main__.SimpleList,
 object]
```

[38]:
```
sil = SortedIntList([5, 10, 15])
super(SortedList, sil)
```

[38]:
```
<super: __main__.SortedList, SortedIntList([5, 10, 15])>
```

[39]:
```
super(SortedList, sil).add(6)
```

[40]:
```
print(sil)
```

```
SortedIntList([5, 10, 15, 6])
```

[41]:
```
super(SortedList, sil).add('7')
```

[42]:
```
print(sil)
```

```
SortedIntList([5, 10, 15, 6, '7'])
```