CS100 Python Introduction to Programming

Lecture 26. Default argument value, Iterator, Generator and Inheritance

Fu Song

School of Information Science and Technology

ShanghaiTech University

Learning Objectives

- **Understand**
  - **Default argument value revisiting**
  - **Iterator**
  - **Generator**
  - **Lazy evaluation**
  - **Inheritance**
  - **User-defined exception**

Mutability

可变的

Mutable vs Immutable?

Depending on whether an object can be changed or not

Default argument value

def funcname(p_1,…,p_n,d_1=v_1,…,d_m=v_m):
    body

- all the parameters occurred at right hand of some parameter having default value must have default values

Important warning: The default value is evaluated **only once**. This makes a difference when the default is a **mutable** object such as a list, dictionary, or instances of most classes.

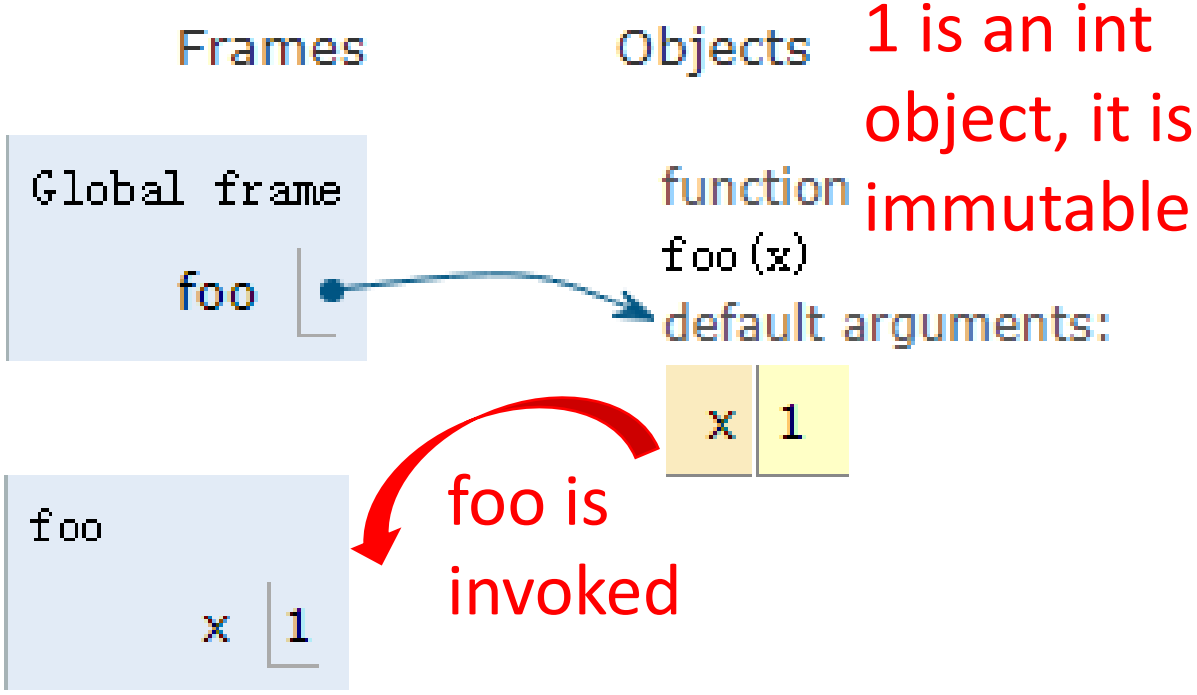
Default argument value

```
def foo(x = 1):
    print(x)
    x = 2

foo()
foo()
foo()
```

Output

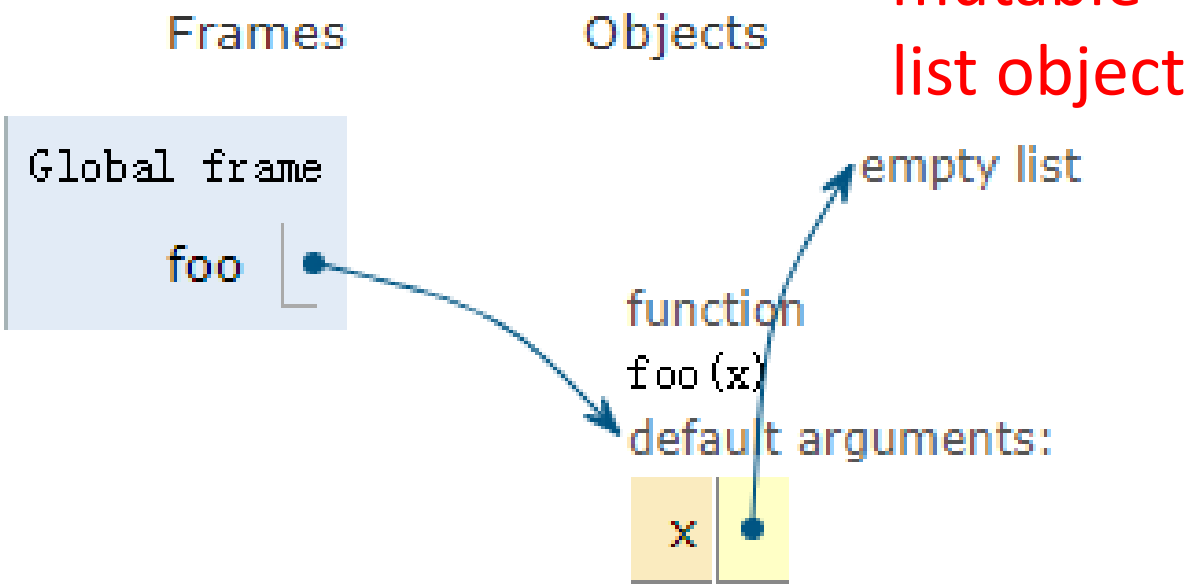
```
1
1
1
>>>
```



Default argument value

```
def foo(x = []):
    print(x)
    x.append(1)

foo()
foo()
foo()
```



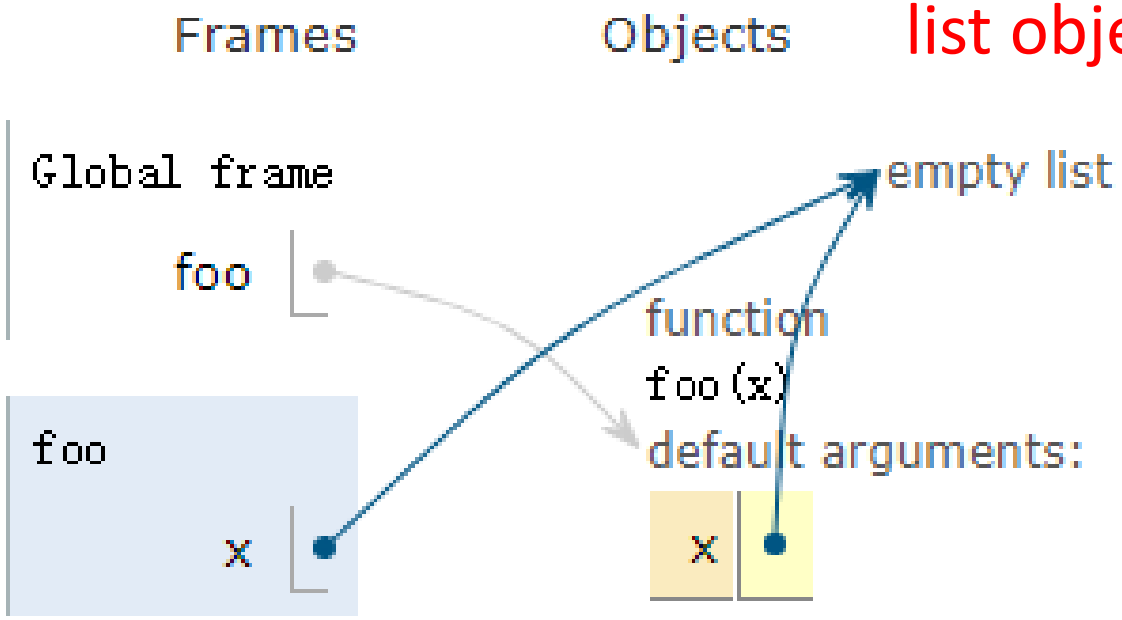
[] is a mutable list object

Default argument value

```
def foo(x = []):
    print(x)
    x.append(1)

foo()
foo()
foo()
```

[] is a mutable list object



foo is invoked at first time

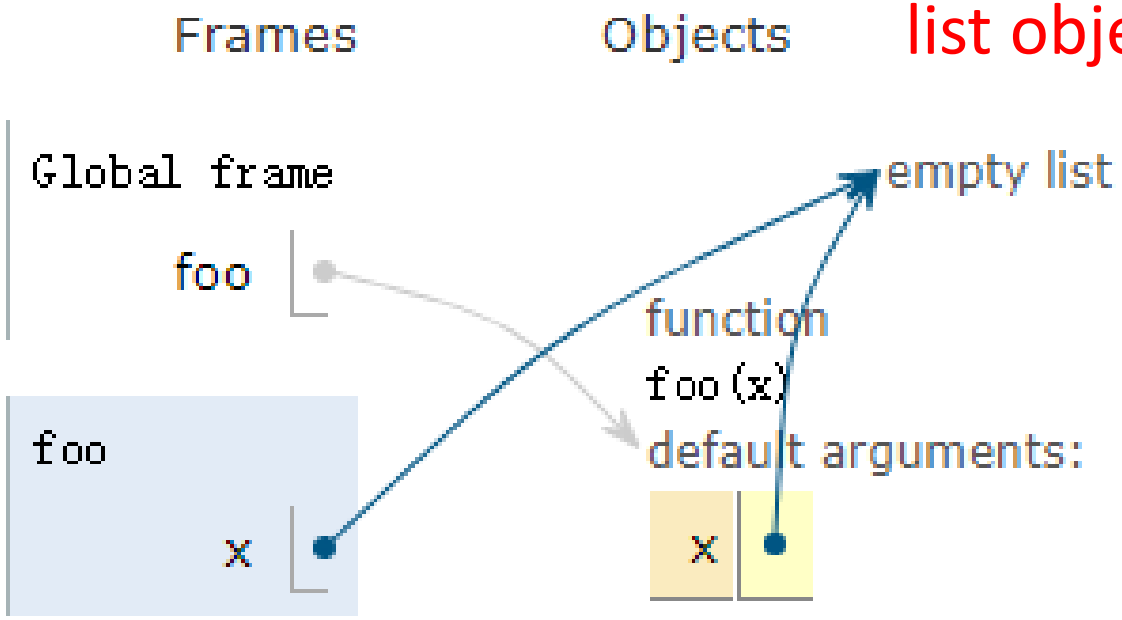
Default argument value

```
def foo(x = []):
    print(x)
    x.append(1)

foo()
foo()
foo()
```

Output []

[] is a mutable list object



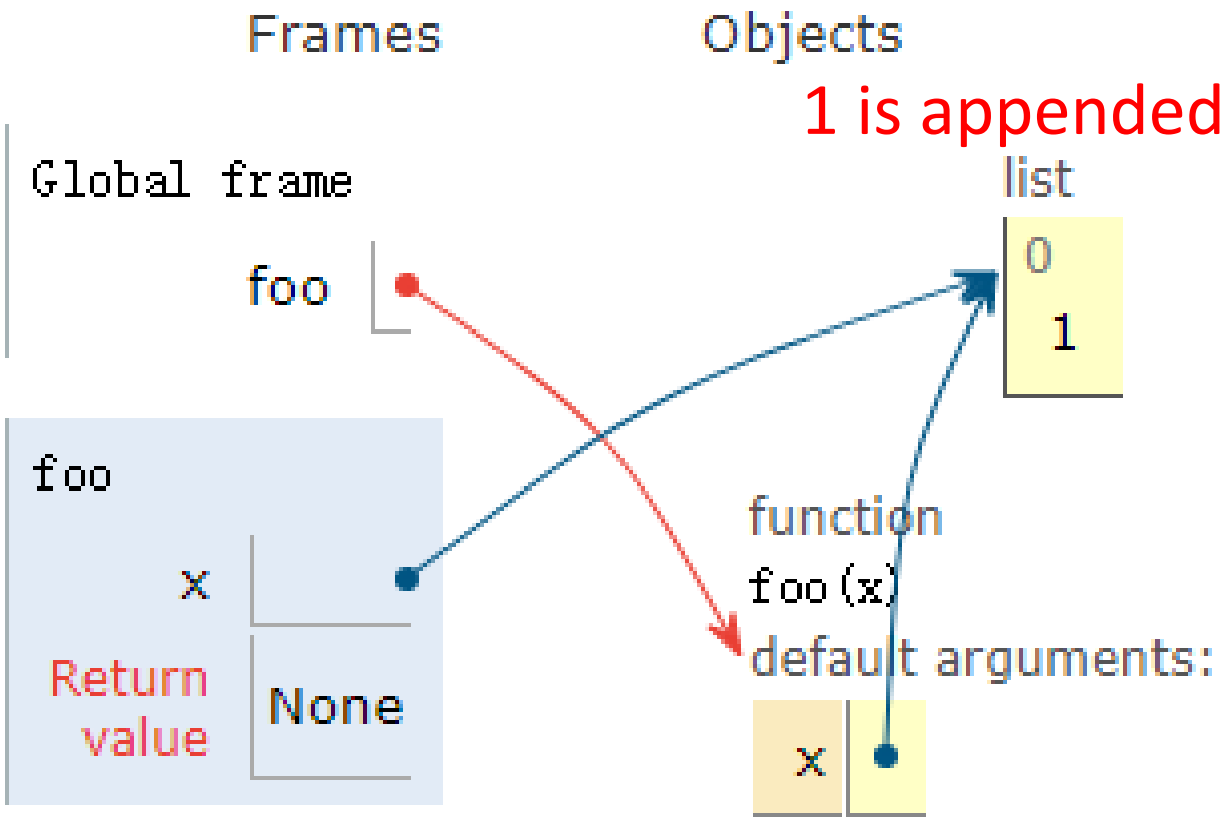
foo is invoked at first time

Default argument value

```
def foo(x = []):
    print(x)
    x.append(1)

foo()
foo()
foo()
```

Output []



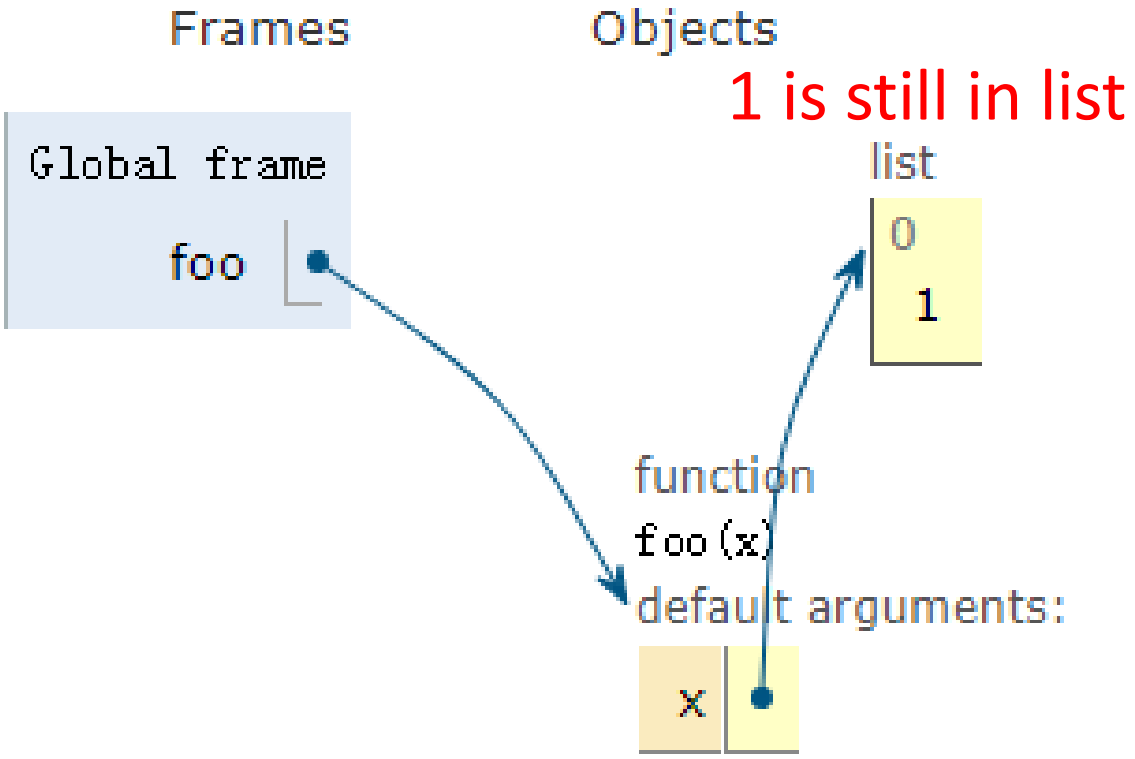
foo is invoked at first time

Default argument value

```
def foo(x = []):
    print(x)
    x.append(1)

foo()
foo()
foo()
```

Output []



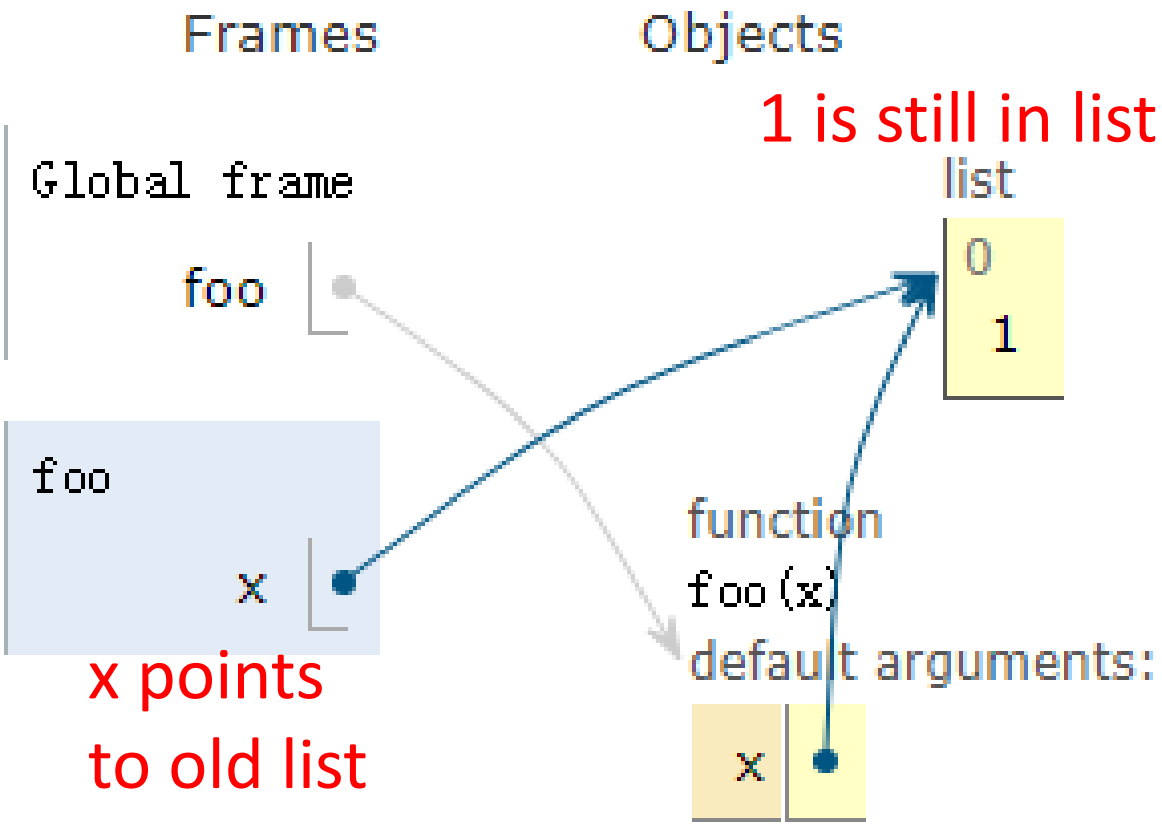
Before second foo is invoked

Default argument value

```
def foo(x = []):
    print(x)
    x.append(1)

foo()
foo()
foo()
```

Output []



foo is invoked at second time

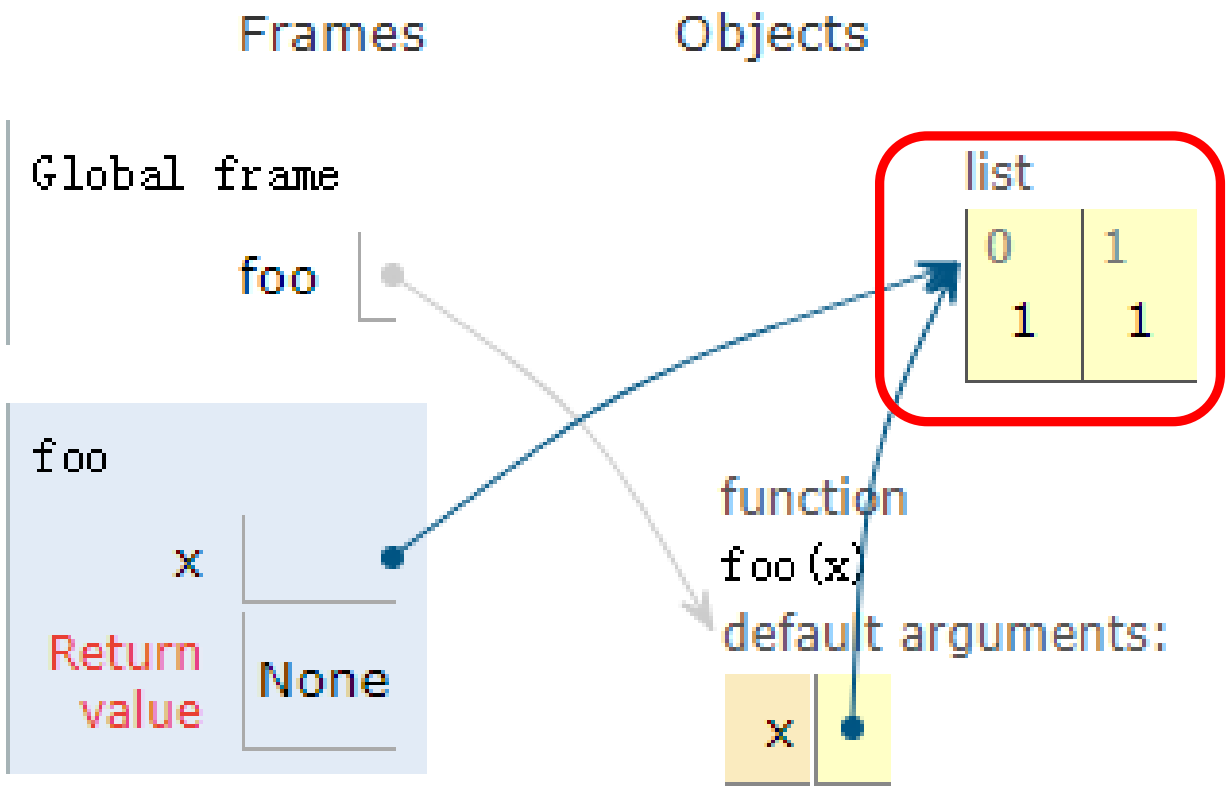
Default argument value

```
def foo(x = []):
    print(x)
    x.append(1)

foo()
foo()
foo()
```

Output

```
[]
[1]
```



foo is invoked at second time

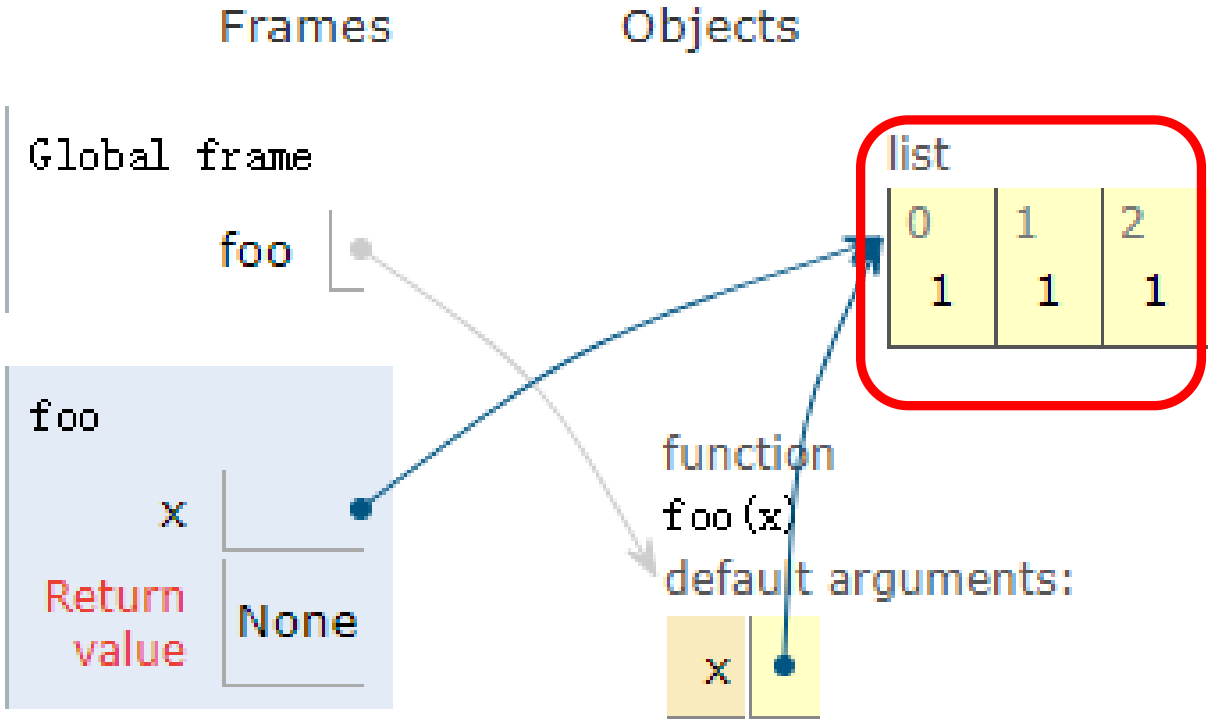
Default argument value

```
def foo(x = []):
    print(x)
    x.append(1)

foo()
foo()
foo()
```

Output

```
[]
[1]
[1,1]
```



foo is invoked at third time

Learning Objectives

- Understand
  - Default argument value revisiting
  - Iterator
  - Generator
  - Lazy evaluation
  - Inheritance
  - User-defined exception

python 哪些类型可迭代?

Iterator

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
```

1. The for statement calls iter() on the container object
2. iter() returns an iterator object that defines __next__() which accesses elements in the container one at a time
3. When there are no more elements, __next__() raises a StopIteration exception which tells for loop to terminate

The iterator protocol

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
next(it)
StopIteration
```

User-defined class supporting iteration

- Override __iter__() method which returns an object with a __next__() method

- If the class defines __next__(), then __iter__() can just return self

User-defined class supporting iteration

```
class Reverse:
    """Iterator for looping over a sequence
backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

User-defined class supporting iteration

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
... print(char)
...
m
a
p
s
```

User-defined class supporting iteration

```
class Counter(object):
    ''' Iterable counter '''

    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        '''Returns itself as an iterator object'''
        return self

    def __next__(self):
        '''Returns the next'''
        if self.current > self.high:
            raise StopIteration
        self.current += 1
        return self.current - 1
```

User-defined class supporting iteration

```
>>> c = Counter(5,8)
>>> print(c)
<Counter.Counter object at 0x0364A530>
>>> next(c)
5
>>> next(c)
6
>>> next(c)
7
>>> next(c)
8
>>> next(c)
Traceback (most recent call last):
……
StopIteration
```

Iterator creation via map

Map(function, iterable,…)

```
>>> x =map(lambda x: x ** 2, range(3))
>>> next(x)
0
>>> next(x)
1
>>> next(x)
4
>>> next(x)
Traceback (most recent call last):
File "<pyshell#31>", line 1, in <module>
next(x)
StopIteration
>>>
```

Iterator creation via Map

Map(function, iterable,…)

```
>>> x =map(lambda x,y: x+y, range(3),range(2,6))
>>> next(x)
2
>>> next(x)
4
>>> next(x)
6
>>> next(x)
Traceback (most recent call last):
File "<pyshell#48>", line 1, in <module>
next(x)
StopIteration
```

Learning Objectives

- Understand
  - Default argument value revisiting
  - Iterator
  - Generator
  - Lazy evaluation
  - Inheritance
  - User-defined exception

Generator

- **Generators are a simple and powerful tool for creating iterators**
- **They are written like regular functions but use the yield statement whenever they want to return data**
- **Each time next() is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed)**

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
print(reverse)
for char in reverse('spam'):
    print(char)
```

Generator

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

print(reverse)
for char in reverse('spam'):
    print(char)
```

Output

```
<function reverse at 0x02DCA150>
m
a
p
s
>>>
```

Generator

```
def foo():
    print("begin")
    for i in range(3):
        print("before yield", i)
        yield i
        print("after yield", i)
    print("end")
```

Generator

```
def foo():
 print("begin")
 for i in range(3):
   print("before yield", i)
   yield i
   print("after yield", i)
 print("end")
```

```
>>> import os
>>> os.chdir("D:\\Test")
>>> from foo import foo
>>> f = foo()
>>> print(f)
<generator object foo at
0x035A7470>
>>> next(f)
begin
before yield 0
0
>>> next(f)
after yield 0
before yield 1
1
```

Generator

```
def foo():
 print("begin")
 for i in range(3):
   print("before yield", i)
   yield i
   print("after yield", i)
 print("end")
```

```
……
>>> next(f)
after yield 1
before yield 2
2
>>> next(f)
after yield 2
end
Traceback (most recent
call last):
File "<pyshell#8>",
line 1, in <module>
next(f)
StopIteration
>>>
```

Generator vs class-based iterator

- Anything that can be done with generators can also be done with class-based iterators
- Generators are more compact as the __iter__() and __next__() methods are created automatically
- Local variables and execution state are automatically saved between calls using generators
- When generators terminate, they automatically raise StopIteration
- These features make it easy to create iterators with no more effort than writing a regular function

Generator Expressions

- **Some simple generators can be coded succinctly as generator expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets**

```
[x**2 for x in range(10)] # list comprehension
```

```
(x**2 for x in range(10)) # generator expression
```

Generator Expressions

```
def reverse(d):
    return (d[i] for i in range(len(d)-1, -1, -1))

print(reverse('spam')
for char in reverse('spam'):
    print(char)
```

```
<generator object reverse.<locals>.<genexpr> at
0x02AC4570>
m
a
p
s
>>>
```

Generator Expressions

```
>>> sum(i*i for i in range(10)) # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec,yvec)) #dot product
260
```

将可迭代对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表.

Generator Expressions vs Generator and List comprehensions

- Generator expressions are simple generators
- Generator expressions are designed for situations where the generator is used right away by an enclosing function
- Generator expressions are more compact but less versatile than full generator definitions
- Generator expressions tend to be more memory friendly than equivalent list comprehensions

Learning Objectives

- Understand
  - Default argument value revisiting
  - Iterator
  - Generator
  - Lazy evaluation
  - Inheritance
  - User-defined exception

Eager evaluation and Lazy evaluation

- **Eager** evaluation (a.k.a. strict evaluation or greedy evaluation): is the evaluation strategy where an expression is evaluated **as soon as it is bound to a variable**
  - used by most traditional programming languages
  - Almost of Python

- **Lazy** evaluation (a.k.a. call-by-need): is an evaluation strategy which **delays** the evaluation of an expression until its value is needed
  - used by most functional programming languages
  - Iterator and generator of Python

```
import time
x = [lambda: "no sleep",
     lambda: time.sleep(10),
     lambda: time.sleep(20)][0]
print(x)
print(x())

x = ["no sleep",
     time.sleep(10),
     time.sleep(20)][0]
print(x)

<function <lambda> at 0x0346A150> # no delay
no sleep # no delay
no sleep # 30s delay
```

Lazy evaluation

Eager evaluation

Lazy evaluation

Advantages

- discard sub-expressions that are not directly linked to the final result of the expression
- reduce the time complexity of an algorithm by discarding the temporary computations and conditionals
- suitable for loading large/infinite data which will be infrequently accessed

Drawbacks

- storing delayed objects for future the evaluation
- sometimes increases space complexity
- difficult to find its performance because it contains delayed objects of expressions before their execution

Lazy evaluation

The range method: returns an iterable range object

- **follows the concept of lazy evaluation**
- **saves the execution time for larger ranges**
- **we never require all the values at a time, so it saves memory consumption as well**

```
>>> x = range(10**10)
>>> x
range(0, 10000000000)
>>> x[10**10-1]
9999999999
>>>
list(x)
......  Long long time
```

Infinite sequence via Lazy evaluation

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fibb = fibonacci()
print(fibonacci)
print(fibb)
for i in range(10):
    print(next(fibb), end=' ')
```

Output

```
<function fibonacci at 0x03902150>
<generator object fibonacci at 0x038F64F0>
0 1 1 2 3 5 8 13 21 34
```

Infinite sequence via Lazy evaluation

```
def odd():
    a = 1
    while True:
        yield a
        a += 2

n = odd()
for i in range(10):
    print(next(n), end=' ')
```

```
def even():
    a = 0
    while True:
        yield a
        a += 2

e = even()
for i in range(10):
    print(next(e), end=' ')
```

Learning Objectives

- Understand
  - Default argument value revisiting
  - Iterator
  - Generator
  - Lazy evaluation
  - Inheritance
  - User-defined exception

Class revisit

```
“class” ClassName”:”
    <statement-1>
    .
    .
    <statement-N>
```

- Class object vs instance object
- Class attribute vs instance attribute
- Private attribute vs public attribute

Inheritance

- **Inheritance is yet another way to reuse code**
- **Other ways:**
  - **Functions**
  - **Classes**
  - **Modules**

Inheritance

```
"class" SubClass "(" BaseClass ")" ":"
    <statement-1>
    .
    .
    <statement-N>
```

- **SubClass** is meant to be more specialized than BaseClass
  – adding new attributes (variables and methods)
- **SubClass** inherits some attributes of BaseClass
- **SubClass** can override inherited methods

Inheritance

- Sub class inherits all public class attributes of the Base class,
- But, does not inherit any private class attributes of Base classes

```
class SubClass(BaseClass):
    y = "SubClass Y"

    @classmethod
    def __private(cls):
        print("Method-1")

    @classmethod
    def public(cls):
        cls.__private()
        BaseClass.public()
        print(cls.__x)

print(SubClass.x)
print(SubClass.y)
SubClass.public()
```

```
class BaseClass:
    x = "BaseClass X"
    y = "BaseClass Y"
    __x = "Private X"

    @classmethod
    def __private(cls):
        print("Method-2")

    @classmethod
    def public(cls):
        print("Method-3")
```

BaseClass X # inherited
SubClass Y # overridden
Method-1
Method-3 # call in base
AttributeError # private

Inheritance

- Sub class inherits all public class attributes of the Base class,
- But, does not inherit any private class attributes of Base classes
- Sub class inherit all public instance methods of the Base class
- But, sub class inherit all public instance variables of the Base class, only if one of the following condition holds
  1. The sub class does not override __init__ method of the base class, (meaning __init__ of the base class is implicitly invoked)
  2. The sub class explicitly invokes __init__ method of the base class in its own __init__ method

如果override会怎样

```python
class BaseClass:
    def __init__(self):
        self.x = "BaseClass X"
        self.y = "BaseClass Y"
        self.__x = "Private X"
    def __private(self):
        print("Method-2")

    def public(self):
        print("Method-3")

class SubClass(BaseClass):
    def __private(self):
        print("Method-1")
    def public(self):
        self.__private()
        BaseClass.public(self)
        print(self.__x)
```

```python
s = SubClass()
print(s.x)
print(s.y)
s.public()
```

子类--private(self)
虽重名但非继承而来

BaseClass X # inherited
BaseClass Y # inherited
Method-1
Method-3
AttributeError ?

```
class BaseClass:
    def __init__(self):
        self.x = "BaseClass X"

class SubClass(BaseClass):
    def __init__(self):
        self.z = "Subclass Z"
        BaseClass.__init__(self)


s = SubClass()
print(s.z)
print(s.x)
```

Subclass Z
BaseClass X

```
class BaseClass:
    def __init__(self):
        self.x = "BaseClass X"

class SubClass(BaseClass):
    def __init__(self):
        self.z = "Subclass Z"

s = SubClass()
print(s.z)
print(s.x)
```

Subclass Z

AttributeError: 'SubClass' object has no attribute 'x'

Inheritance

- Sub class inherits all public class attributes of the Base class,
- But, does not inherit any private class attributes of Base classes
- Sub class inherit all public instance methods of the Base class
- But, sub class inherit all public instance variables of the Base class, only if one of the following condition holds
  1. The sub class does not override __init__ method of the base class, (meaning __init__ of the base class is implicitly invoked)
  2. The sub class explicitly invokes __init__ method of the base class in its own __init__ method
- New/overridden method cannot access private attributes of the base class
- But, inherited methods can access private attributes of the base class

```
class BaseClass:
    def __init__(self):
        self.__x = "Private X“

    def __private(self):
        print("Private 1")
        print(self.__x)

class SubClass(BaseClass):
    def __private(self):
        print("Private 2")

    def public(self):
        self.__private()
        print(self.__x)
s = SubClass()
s.public()
```

Private 2
AttributeError: 'SubClass' object has no attribute '_SubClass__x

New/overridden method cannot access private attributes of the base class

```
class BaseClass:
    def __init__(self):
        self.__x = "Private X"

    def __private(self):
        print("Private 1")
        print(self.__x)

    def public(self):
        self.__private()
        print(self.__x)

class SubClass(BaseClass):
    def __private(self):
        print("Private 2")
s = SubClass()
s.public()
```

```
Private 1
Private X
Private X
```

inherited methods can access private attributes of the base class

和下页区别的原因

```
class BaseClass:
    def __init__(self):
        self.__x = "Private X“

    def private(self):
        print("Private 1")
        print(self.__x)

    def public(self):
        self.private()
        print(self.__x)

class SubClass(BaseClass):
    def private(self):
        print("Private 2")
s = SubClass()
s.public()
```

Private 2
Private X

inherited methods will first search in sub class

Multiple Inheritance

```
“class” SubClass “(” Base1, Base2,…,Basen “)” ”:”
   <statement-1>
   .
   .
   <statement-N>
```

- Python supports multiple inheritance
- SubClass is a specialization of all base classes

Class inheritance graphs

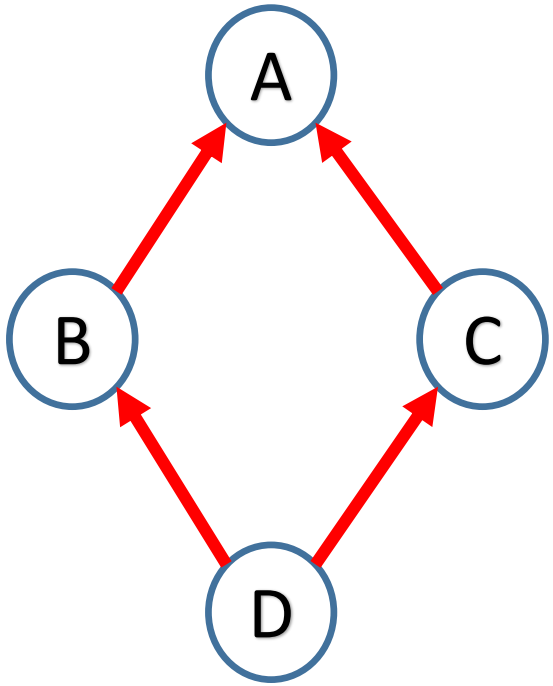
- **Each class is a node**
- **If class A inherits from class B, then there is an edge (A,B)**

```
class A:
    def get(self):
        return "A"

class B(A):
    def get(self):
        return “B"

class C(A):
    pass

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



Method Resolution Order (MRO)

- **Which name is inherited when several base classes have this name?**

```
class A:
    def get(self):
        return "A"

class B(A):
    def get(self):
        return “B"

class C(A):
    pass

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

```
s = D()
print(s.get())
```

Which is printed?

C3 linearization

- C3 linearization is used to maintain Monotonicity and Local precedence ordering

- Monotonicity: If Base1 > Base2 in the linearization of class A, then Base1 > Base2 in the linearization of any subclass of class A

- Local precedence ordering : If Sub(Base1,Base2,Base3), then the local precedence ordering for Sub is

  Base1>Base2>Base3

C3 linearization

class sub(B_1,...,B_n):

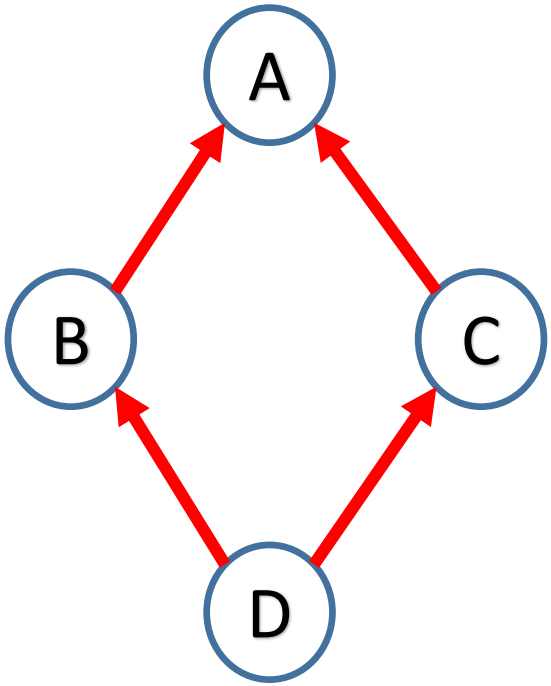
……

MRO: **$L(Sub(B_1 \dots B_n)) = sub + merge(L(B_1) \dots L(B_n), B_1 \dots B_n)$**

Merge: repeat the following operations until all the classes are removed or halt (raise an exception TypeError)

1. take the head of the first list, i.e, $L(B_1)[0]$
2. if this head is not in the tail (all elements of a list except the first) of any of the other lists, add it to the linearization of Sub and remove it from the lists in the merge
3. otherwise look at the head of the next list and take it, apply step 2, if no next list, then halt and TypeError

$$L(Sub(B_1 \dots B_n)) = Sub + merge(L(B_1) \dots L(B_n), B_1 \dots B_N)$$



$L(A) = [A]$

$L(B) = [B] + merge(L(A), [A])$

$= [B] + merge([A], [A])$

$= [B, A]$

$L(C) = [C] + merge(L(A), [A])$

$= [C] + merge([A], [A])$

$= [C, A]$

$L(D) = [D] + merge(L(C), L(B), [C, B])$

$= [D] + merge([C, A], [B, A], [C, B])$

$= [D,C] + merge([A], [B, A], [B])$

$= [D,C, B] + merge([A], [A])$

$= [D,C, B, A]$

Method Resolution Order (MRO)

- **Which name is used when several base classes have this name (or inherited from their base classes)?**

```
class A:
    def get(self):
        return "A"

class B(A):
    def get(self):
        return “B"

class C(A):
    pass

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

```
s = D()
print(s.get())
```

MRO: [D,C, B, A]

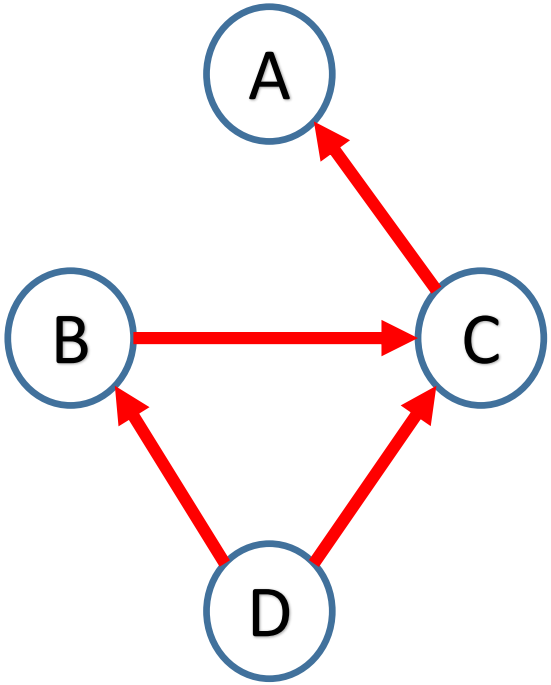
Output: B

Method Resolution Order (MRO)

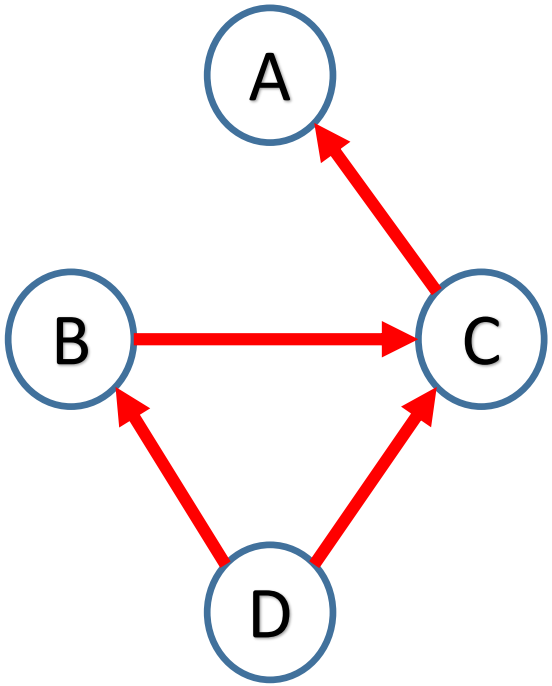
- **Which name is used when several base classes have this name (or inherited from their base classes)?**

```
class A:
    def get(self):
        return "A"
class C(A):
    pass
class B(C):
    def get(self):
    return "B“
class D(C,B):
    pass
```

```
s = D()
print(s.get())
```



$$L(Sub(B_1 \dots B_n)) = Sub + merge(L(B_1) \dots L(B_n), B_1 \dots B_N)$$



$L(A) = [A]$

$L(C) = [C] + merge(L(A), [A])$

$= [C] + merge([A], [A])$

$= [C, A]$

$L(B) = [B] + merge(L(C), [C])$

$= [B] + merge([C, A], [C])$

$= [B, C] + merge([A])$

$= [B, C, A]$

$L(D) = [D] + merge(L(C), L(B), [C,B])$

$= [D] + merge([C, A],[B, C,A],[C,B])$

>>>TypeError

Method Resolution Order (MRO)

- **Which name is used when several base classes have this name (or inherited from their base classes)?**

```
class A:
    def get(self):
        return "A"
class C(A):
    pass
class B(C):
    def get(self):
    return "B“
class D(C,B):
    pass
```

```
s = D()
print(s.get())
```

TypeError: Cannot create a consistent method resolution order (MRO) for bases C, B

Warning

- **Don’t write classes have complex inheritance relation**

- **If you cannot sure the MRO of your classes, check using Myclass.__mro__, e.g.**

```
print(D.__mro__)
```

Learning Objectives

- Understand
  - Default argument value revisiting
  - Iterator
  - Generator
  - Lazy evaluation
  - Inheritance
  - User-defined exception

Exception hierarchy: Built-in

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

User-defined exception

- User-defined exceptions can be created by defining a new exception class, typically be derived from the Exception class, either directly or indirectly
- Exception classes can be defined which do anything, but are usually kept simple, often only offering a number of attributes that allow information about the error
- When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions

```
class Error(Exception):
    """Base class for your module."""
    pass

class InputError(Error):
    """Exception error for input."""
    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg

def checkSyntax(s):
    return False

try:
    x = input("Input your expr:")
    if checkSyntax(x) == False:
        raise InputError(x, “Error Information")
    except InputError as e:
        print(e.msg, "in", x)
```

Recap

- Understand
  - Default argument value revisiting
  - Iterator
  - Generator
  - Lazy evaluation
  - Inheritance
  - User-defined exception