ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Lesson 8

ILAI (M1) @ LAAI I.C. @ LM AI

21 October 2024

Michael Lodi

Department of Computer Science and Engineering

These slides draw very heavily from Simone Martini's slides.

One-slide recap of lesson 7

A subclass inherits from a superclass.

An instance of a subclass is also an instance of its superclasses (`isinstance()` vs `type()`)

All Python classes (types) inherit from the superclass `object`

Subclasses inherit attributes and methods unless they are redefined (overridden). Subclasses can add attributes and methods (subclass has «more information» than superclass)

`super()` allows us to access methods of the superclass that have been overridden. Think as «`self` seen in its immediate superclass»

Dynamic method lookup: unlike functions, the actual method to be executed is determined at runtime, depending on the actual type (class) of the object on which the method is invoked.

Late binding of `self`: calling a yet-to-be-implemented method, *delegating* subclasses to implement it (realizing the idea of interface)

In Python, a class can inherit from more than one class (multiple inheritance)

The order of precedence (to decide which `super()` and which methods to inherit) is defined by an algorithm: the `MRO`. The order is not always defined (i.e. not legal inheritance)

No tuple comprehension. Comprehension-like expression in ( ) is a generator, a «potential sequence». Calling `next()` will generate the next element of a sequence. When ended, the generator is exhausted. We can do (once) a `for` loop over a generator.

Explicit form of generator: Any `def` whose body contains *(at least) one* `yield`

Iterators: most general case. «Something on which I can do a for loop.»

Each class that defines `__iter__` and `__next__`

Next lecture for ILAI

Tomorrow, 22 October 2024

09:15 - 12:00 Room 0.5 (Covering prof Chesani)

LAST LECTURE!

Who to contact

For questions on exercises, on Python, tutoring/mentoring

1. *contact* **federico.ruggeri6@unibo.it**
   or
   mohammadrez.hossein3@unibo.it
2. *if you still have questions (eg. Theoretical), contact me: michael.lodi@unibo.it*

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Early exam - 8th November, 15:00 - EVERYONE IS WELCOME*

1. With a @studio.unibo.it account - mandatory
2. an account on the Ingegneria cluster: https://remo.ing.unibo.it/app/student/infoy (you need the step 1 account to generate the ing account)
3. **At the latest 1 week before** access (with the step 1 account) the website https://eol.unibo.it/

Register for the exam on the Unibo app or Almaesami - **PREFERRED**

- https://almaesami.unibo.it/almaesami/welcome.htm

Only if you are not able to register on app/Almaesami

https://forms.office.com/e/RaYAWQMBP2 (using your @studio account only)

If you have already filled the form but become able to register on AlmaEsami, do so.

Between 7 October and 4 November (included). Late requests will not be accepted.

Essential that you don't skip other lectures of other courses to study for this

exceptions

errors (exceptions) come in various "kinds"

```
print(1/0)
```

produces an error:

`ZeroDivisionError:` division by zero

```
L=[1,2,3]
print(L[10])
```

produces *another* error:

`IndexError:` list index out of range

Terminology

raising an exception (an error)

Why "exception" and not "error"

"Error" is too general:

syntactic:

```
if 'a' in 'ciao'
    print('found')
```

semantic:

```
if a%2 == 0:
    print(a, 'is odd')
```

dynamic:

```
print(1/0) # this raises an exception
```

We may handle an exception

We may *handle* an exception with a `try/except` command

```
try:
    print(1/0)
except ZeroDivisionError:
    print('OK!')
print('after')
```

The `ZeroDivisionError` exception is *caught* by the except clause

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

We may handle an exception

We may *handle* an exception with a `try/except` command

```
try:
        print(1/0)
except IndexError:
        print('OK!')
print('after')
```

The `ZeroDivisionError` exception makes the program terminate, because is not caught by the except

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

`try/except` command

```
try:
    <block>
except <exception>:
    <handlerblock>
<after>
```

`<block>` is executed.

If no exception occurs, then execution proceeds to <after>

If `<exception>` occurs, `<handlerblock>` is executed; then `<after>`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

`try/except` command

an

```
    except:
        <handlerblock>
```

catches *all* exceptions

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

`try/except` command

the portion of <block> evaluated before the exception maintains its effects

```
try:
    X=10
    Y=100/0
    X=100
except:
   Y=200
#here X is 10 and Y is 200
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Other Languages: LBYL

Look before you leap

```
D = {'apple':3, 'pear':4}
if 'banana' in D:
    print(D['banana'])
else:
    print('No bananas')
```

Python: EAFP

Easier to ask for forgiveness than permission

```
D = {'apple':3, 'pear':4}
try:
    print(D['banana'])
except KeyError:
    print('No bananas')
```

A programming example

We input *integers* and print their square, until 'stop' is read

If a non number is read, signal the error and proceed with next input

A programming example

We input *integers* and print their square, until 'stop' is read

If a non integer is read, signal the error and proceed with next input

```
reply = input('Enter text (or "stop"): ')
while reply != 'stop':
    if not reply.isdigit():
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
    reply = input('Enter text (or "stop"): ')
print('Bye')
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

A programming example

We input *integers* and print their square, until 'stop' is read

If a non integer is read, signal the error and proceed with next input

```
reply = input('Enter text (or "stop"): ')
while reply != 'stop':
    try:
        print(int(reply) ** 2)
    except ValueError:
        print('Bad!' * 8)
    reply = input('Enter text (or "stop"): ')
print('Bye')
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

A programming example

Does Python have a <do…while> or <repeat…until>?

No. The idiomatic coding in Python would be

```
while True:
    reply = input('Enter text (or "stop"): ')
    if reply=='stop':
        break
    try:
        print(int(reply) ** 2)
    except:
        print('Bad!' * 8)
print('Bye')
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

A programming example

Does Python have a <`do…while`> or <`repeat…until`>?

Asking user for valid input:

https://stackoverflow.com/questions/23294658/asking-the-user-for-input-until-they-give-a-valid-response

I like also the «while … is None» solution:

```
x = None
while x is None: # or the more obscure «while not x:»
    try:
        x = int(input('Give me an integer: '))
    except ValueError:
        print('You did not gave me an integer')
print("Thank you for the integer", x)
```

`try/except` command

```
try:
    <block>                    # run this first
except name1:
    <block1>                   # run if name1 is raised in try block
except (name2, name3):         # mandatory parentheses
    <block2>                   # run if any of these exceptions raised in try
except name4 as name:
    <block4>                   # run if name4 is raised, assign instance to name
except:
    <block>                    # run for all other exceptions raised
else:
    <block>                    # run if no exception was raised in try block
```

`try/except` command

```
try:
    <block>                  # run this first
except name1:
    <block>                  # run if name1 is raised in try block
except (name2, name3):
    <block>                  # run if any of these exceptions raised in try
except name4 as name:
    <block>                  # run if name4 is raised, assign instance to name
except:
    <block>                  # run for all other exceptions raised
else:
    <block>                  # run if no exception was raised in try block
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

`try/except` command

```
try:
    <block>                              # run this first
except name4 as name:
    <block>                # run if name4 is raised, assign instance to name
```

Exceptions usually come *also* with a value

E.g. the standard `ZeroDivisionError` comes with the string value `"division by zero"`

```
try:
    print(1/0)
except ZeroDivisionError as MyName:
    print(MyName, type(MyName))
    print('OK')
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

`try/except` command

Exceptions usually come *also* with a value

```
>>> '2' + 2
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    '2' + 2
TypeError: can only concatenate str (not "int") to str

>>> 2+'2'
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    2+'2'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

`try/except` command

```
try:
    <block>                    # run this first
except name1:
    <block>                    # run if name1 is raised in try block
except (name2, name3):
    <block>                    # run if any of these exceptions raised in try
except name4 as name:
    <block>                    # run if name4 is raised, assign instance to name
except:
    <block>                    # run for all other exceptions raised
else:
    <elseblock>                # run if no exception was raised in try block
<after>
```

Why having an `else` clause? <elseblock> is run if no *exception is raised in try block*

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

`try/except` command

```
try:
    <block>                    # run this first
except name1:
    <block>                    # run if name1 is raised in try block
except (name2, name3):
    <block>                    # run if any of these exceptions raised in try
except name4 as name:
    <block>              # run if name4 is raised, assign instance to name
except:
    <block>                    # run for all other exceptions raised
else:
    <elseblock>                # run if no exception was raised in try block
<after>
```

Why having an `else` clause? <elseblock> is run if no *exception is raised in try block*

Did we get at <after> because the try failed or not?

`try/except` command

```
try:
    <block>
except . . .:
    . . .
else:
    <elseblock>
<after>
```

Is very much like:

try/except command

```
try:
    <block>
except . . .:
    . . .
else:
    <elseblock>
<after>
```

Is very much like:

```
try:
    <block>
    <elseblock>
except . . .:
    . . .
<after>
```

`try/except` command

```
try:
    <block>
except . . .:
    . . .
else:
    <elseblock>
<after>
```

Is very much like:

```
try:
    <block>
    <elseblock>
except . . .:
    . . .
<after>
```

But what if `<elseblock>` raises an exception??

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

the finally clause

```
try:
    <block>                    # run this first
except name1:
    <block>
. . .
else:
    <block>                    # run if no exception was raised in try block
finally:
    <finalblock>               # run in any case before leaving the try
```

<finalblock> is executed in any case, as final code, either if exceptions are or are not raised, or are handled or non handled

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

the finally clause

<finalblock> is executed in any case, as final code,
either if exceptions are or are not raised,
or are handled or non handled

```
try:
    print(1/0)
except IndexError:
    print("Bad index") #or even a return
finally:
    print('I am executed in any case')
```

```
>>> %Run
I am executed in any case
Traceback (most recent call last):
  File "", line 2, in <module>
    print(1/0)
ZeroDivisionError: division by zero
```

the finally clause

```
finally:
    <finalblock>                    # run in any case before leaving the try
```

`<finalblock>` *is executed in any case, as final code,*
either if exceptions are or are not raised,
or are handled or non handled

We use `finally` *to release resources:*
disconnect from network, close/release files or locks, etc.

Exceptions and the frame stack

The normal programming situation:

the exception *will not happen directly in the try* block

but *in a function/method called from the inside* the try block

```
try:
    f()
except <excpname>:
    print('Excpname caught')
```

`f()` may call `g()`, which may call `h()`

and only at that point `Excpname` is raised

Exceptions and the frame stack

```
def f():
    a=0
      g()
    return 1
def g():
    h()
    return 2
def h():
    c=0
    return 1/0  #ZeroDivisionError raised

try:
    print(f())
except ZeroDivisionError:
    print('ZeroDivisionError caught')
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Exceptions and the frame stack

Exceptions are propagated through the stack

Frames are popped out of the stack

if their function does not handle the exception

The code remaining in those functions is not executed

If an exception is never caught, the exception is passed to the standard exception handler of the main:

the program terminates and the exception is printed

We may explicitly raise an exception

The command

`raise` <exception_instance>

raises the exception

```
print('first')
raise IndexError
print('second')
```

We may explicitly raise exceptions

```
class Count:
    def __init__(self, start=0):
        if type(start) != int:
            raise TypeError
        if start < 0:
            raise ValueError
        self.current_count = start
    def inc(self):
        self.current_count += 1
```

Some built-in exceptions

ZeroDivisionError

IndexError

KeyError

TypeError

ValueError

NameError

OSError

User defined exceptions

An exception is an instance of (a subclass of) `Exception`

```
class MyExcp(Exception):
    pass
...
try:
    raise MyExcp    #short for MyExcp()
except MyExcp:
    print("caught")
```

Meaningless toy example: rasing an exp and immediatly catching it

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

`except. . .as` clause

We may add a value to the instance, so that we may use that value in the handler

```
class MyExcp(Exception):
    pass
. . .
try:
    raise MyExcp(10)
except MyExcp as X:
    print("caught", X)
```

We may explicitly raise exceptions

```
class Count:
    def __init__(self, start=0):
        if type(start) != int:
            raise TypeError(str(start)+" not int")
        if start < 0:
            raise ValueError(str(start)+" not >=0")
        self.current_count = start
    def inc(self):
        self.current_count += 1
```

`except. . .as` clause

In more details:

```
except E as X
```

binds to `X` the *instance* of `E` that is raised

In `Exception` there are predefined attributes/methods

```
__str__
__repr__
args
```

`except. . .as` clause

```
class A(Exception):
    pass
try:
    raise A(10)
except A as x:
    print(x.__str__())
    print(x.__repr__())
    print(x.args)
```

`except. . .as` clause

```
class A(Exception):
    pass
try:
    raise A(10)
except A as x:
    print(x.__str__())      # 10
    #print(x)               #equivalent
    print(x.__repr__())     # A(10)
    print(x.args)           # (10,)
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

`except. . .as` clause

Of course we may override them:

```
class A(Exception):
    def __str__(self):
        return super().__str__+'ML'
try:
    raise A(10)
except A as x:
    print(x)                    # 10ML
    print(x.__repr__())         # A(10)
    print(x.args)               # (10,)
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

`except. . .as` clause

In an `except. . .as` *`name`* clause,

`name` is local to the handler and is destroyed when the handler terminates

```
try:
    raise MyExcp(10)
except MyExcp as X:
    print("caught", X)
print(X)   # fails
```

`except. . .as` clause

In an `except. . .as name` clause,

name is local to the handler and is destroyed when the handler terminates. **Very strange and subtle detail (linked with garbage collection)**

```
X = 100
try:
    raise MyExcp(10)
except MyExcp as X:
    print("caught", X)
print(X)# error: X del from current scope
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

User defined exceptions

The argument of `raise` is any reference to an exception object

```
class MyExcp(Exception):
    pass
P=(MyExcp, IndexError)
try:
    raise P[0]
except MyExcp:
    print("caught")
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

User defined exceptions

The argument of `except` is *any* exception class

```
class MyExcp(Exception):
    pass
try:
    raise MyExcp
except Exception:    #same as except:
    print("caught")
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Subclasses of user defined exceptions

A clause

```
except <exceptionclass>
```

will catch all exceptions which are instances of `<exceptionclass>`

(therefore all instances of its subclasses)

Hierarchies of user defined exceptions

A clause

```
      except <exceptionclass>
```

will catch all exceptions which are instances of `<exceptionclass>` **(therefore all instances of its subclasses)**

```
class MyExcp(Exception): pass
class MySpec1(MyExcp): pass
class MySpec2(MyExcp): pass

for e in (MyExcp(0), MySpec1(1), MySpec2(2)):
    try:
        raise e
    except MyExcp as X:
        print('caught ', X)
```

Hierarchies of user defined exceptions

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

for cls in [B, C, D]:
   try:
      raise cls()
   except D:
      print("D")
   except C:
      print("C")
   except B:
      print("B")
```

Will print B, C, D in this order

Hierarchies of user defined exceptions

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

for cls in [B, C, D]:
   try:
      raise cls()
    except B:
      print("B")
    except D:
      print("D")
   except C:
      print("C")
```

Will print B, B, B

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Programming with exceptions: 1

Compute the product of a sequence of integers

(the math module has math.prod, only from Python 3.8)

Iteration, 1

```
def prod(S):
    res=1
    for e in S:
        res *=e
    return res
```

Programming with exceptions

Compute the product of a sequence of integers
(the math module has math.prod, only from Python 3.8)

Iteration, 1

```
def prod(S):
    res=1
    for e in S:
        res *=e
    return res
```

What about stopping when a zero is found?

Programming with exceptions

Compute the product of a sequence of integers
(the math module has math.prod, only from Python 3.8)

Iteration, 2

```
def prod_z(S):
    res=1
    for e in S:
        if e==0:
            return 0
        res *= e
    return res
```

return will break the iteration and terminate the function

Programming with exceptions

Compute the product of a sequence of integers
now use recursion, instead of iteration

Recursion, 1

```
def recprod(S):
    if len(S)==0:
        return 1
    else:
        return S[0]*recprod(S[1:])
```

What about stopping when a zero is found?

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Programming with exceptions

Compute the product of a sequence of integers

now use recursion, instead of iteration

Recursion, 1

```
def recprod(S):
    if len(S)==0:
        return 1
    else:
        if S[0]==0:
            return 0
        return S[0]*recprod(S[1:])
```

is correct but not quite the same as the iterative "return 0"
This will stop further recursive calls, but will be multiplied with the "previous" elements of S

we must *propagate that zero along the chain of the recursive calls*

Programming with exceptions

Recursion, 2

```
class Zero(Exception): pass
def recprod(S):
    def aux(S):
        if len(S)==0:
            return 1
        else:
            if S[0]==0:
                raise Zero
            else:
                return S[0]*aux(S[1:])
    try:
        res=aux(S)
    except Zero:
        return 0
    else:
        return res
```

Programming with exceptions: 2

Breaking out multiple nested loops

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Programming with exceptions: 2

```
class Exitloop(Exception): pass
try:
    while True:
        while True:
            for i in range(10):
                if i > 3:
                    raise Exitloop  # break doesn't do
                print("loop3: ", i)
            print("loop2")
        print("loop1")
except Exitloop:
    print('caught')
```

Prints:
loop3: 0
loop3: 1
loop3: 2
loop3: 3
caught

How much does a program cost?

How much does a program cost?

(Time) complexity of a program/function:

time needed to run it on data D
as a function of the *size* of D

instead of actual time,
count the number of elementary steps

if elementary steps are all constant-time ops,
the estimate is *proportional* to the actual time

How much does a program cost?

(Time) complexity of a program/function:

Let S be a sequence of positive integers

```
m = 0
for e in S:
    if e > m:
        m = e
```

- dimension of S: number *n=len(S)* of elements
- if assignment, single iteration handling, guard evaluation are all constant-time ops, global evaluation is proportional to n (linear time)

How much does a program cost?

```
def maxi(S):
    m=0
    for e in S:
        if e>m:
            m=e
    return m

def maxind(S):
    res=[]
    for i in range(len(S)):
        if S[i]==maxi(S):
            res+=[i]
    return res
```

- dimension of S: number *n=len(S)* of elements
- if assignment, single iteration handling, guard evaluation are all constant-time ops,
  global evaluation of maxind is proportional to n^2 (quadratic time)

How much does a program cost? much better

```
def maxi(S):
    max=0
    for e in S:
        if e>max:
            max=e
    return max

def maxind(S):
    m=maxi(S)
    res=[]
    for i in range(len(S)):
        if S[i]==m:
            res+=[i]
    return res
```

- dimension of S: number *n=len(S)* of elements
- if assignment, single iteration handling, guard evaluation are all constant-time ops,
  global evaluation is proportional to n (linear time)

How much does *an operation* cost?

If

- assignment
- single iteration handling
- guard evaluation

- arithmetical operations
- access to data structures
- . . .

are all constant-time ops…

How much does *an operation* cost?

If

- assignment
- single iteration handling
- guard evaluation

- arithmetical operations
- access to data structures
- . . .

are all constant-time ops…

Are they, in Python ?

How much does *an operation* cost?

Why Python looks suspicious:

Integers in C/Java/Pascal/Fortran etc.

one-to-one correspondence

data type of the language (integer)

internal representation on the
fixed-sized words of machine

minimum and *maximum* representable integer

How much does *an operation* cost?

ARRAYS in C/Java/Pascal/Fortran etc.

simple sequences of values of basic types (int/float)

fixed size

straightforward mapping between

array indexes and memory addresses

Machine word is 4 bytes (32 bits)

1-dimensional array A of 10 integers, indexed from 0 to 9

How much does *an operation* cost?

ARRAYS in C/Java/Pascal/Fortran etc.

Suppose a machine word is 4 bytes (32 bits)

1-dimensional array A: 10 integers
indexed from 0 to 9

stored at address b

A[0] is at address b
A[1] at at address b+4
A[2] at at address b+8
…
A[i] at at address b+4*i

How much does *an operation* cost?

ARRAYS in C/Java/Pascal/Fortran etc.

Suppose a machine word is 4 bytes (32 bits)

1-dimensional array A: 10 integers
indexed from 0 to 9

stored at address b

A[0] is at address b
A[1] at at address b+4
A[2] at at address b+8
…
A[i] at at address b+4*i

simple computation
from index to address
b+4*i
via b and 4 (the *stride*)

this formula is known statically

How much does *an operation* cost?

ARRAYS in C/Java/Pascal/Fortran etc.

Multiple dimension array B 3x3x2

B[i,j,k]

formula slightly more complicated

- is B stored by *rows* (C,Java) or by columns (Fortran)?
- generalised notion of stride: (24,8,4)

the formula is known statically

b+i*24+j*8 +k*4

via b and the stride (24,8,4)

How much does *an operation* cost?

In C/Java/Pascal/Fortran etc.

straightforward mapping between

language values and machine addresses

In Python:

access is always *indirect*

name refers to a descriptor

which refers to a data structure

which contains the value(s)

Flexibility is paid by space and indirection

How much does *an operation* cost?

Why Python is different:

- integers of arbitrary size
- variable length data structures (lists, dicts)

while the lower-level machines have

- fixed-size memory words
- fixed correspondence
  between addresses and memory words

How much does *an operation* cost?

Main distinction

constant-time operations

variable-time operations

the time depends on the *size* of the args

How much does *an operation* cost?

How Python data is represented on a lower-level, fixed-memory word machine ?

One Python's int doesn't fit in a single memory word

del L[i] involves the "recomputation of the indexes"

Reference implementation

CPython

the most common Python implementation

written in C

simple parser (of LL type)

compiler to Python bytecode

interpreter of Python bytecode

run-time support (eg. garbage collector)

The underlying machine

Organised in fixed size words

in binary: 32 bits

hence 2^{32} different possible numbers

We access any word in constant time through its *(memory) address*

For simplicity of exposition:

examples *in decimal*

assuming words of 8 decimal digits

Python int

How are int values represented?

They cannot be mapped directly into the "integers" of the underlying machine!

Python int

How are int values represented?

They cannot be mapped directly into the "integers" of the underlying machine!

objects

garbage collected

arbitrary size

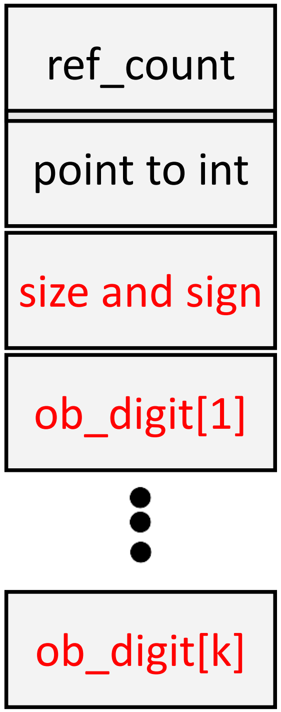
Python int

A *single* Python int

ref_count	for garbage collection
point to int	pointer to the description of the class int
size and sign	sign and how many ob_digit needed
ob_digit[1]	first chunk of the number (little endian)
⋮	
ob_digit[k]	last chunk of the number (little endian)

Python int

A *single* Python int



Let's encode

-123456789101112131415

Python int

A *single* Python int

Let's encode

-123456789101112131415

divide it in chunks of 8 decimal digits

(in reality: 30 bits)

starting from right (*little endian)*

Python int

A *single* Python int

Let's encode

-123456789101112131415

divide it in chunks of 8 decimal digits

(in reality: 30 bits)

starting from right (*little endian)*

- 12345 67891011 12131415

Python int

A *single* Python int

Let's encode

-123456789101112131415

divide it in chunks of 8 decimal digits

(in reality: 30 bits)

starting from right (*little endian)*

- 12345 67891011 12131415

3 chunks, negative sign

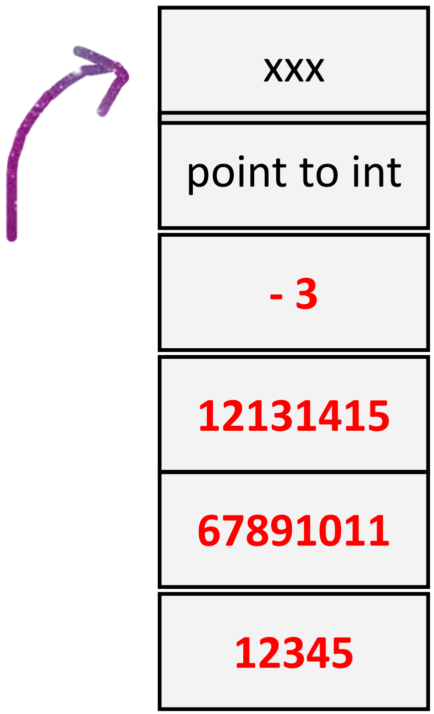
12131415

67891011

12345

Python int

A *single* Python int



Let's encode

-123456789101112131415

divide it in chunks of 8 decimal digits

(in reality: 30 bits)

starting from right (*little endian)*

\- 12345 67891011 12131415

3 chunks, negative sign

12131415

67891011

12345

Python int

Any time an `int` is used

an object of that shape is allocated in the memory

Python int

Any time an int is used

an object of that shape is allocated in the memory

Optimisations: (1)

integers in the range [-2^{30} , 2^{30}] are allocated as

underlying arithmetic on ob_digit[1]

is used if possible

Python int

Any time an int is used

an object of that shape is allocated in the memory

Optimisations: (2)

small integers in the range [-5 , 256]
are pre-allocated at initialisation time

any such object is never duplicated

we may check this in Python, through `id(…)`

Python list

```
[1,2,3]
```

```
[(1,2,3), 3, 'bologna']
```

are both lists of length 3

Python list

`[1,3,2]`

`[(1,2,3), 3, 'bologna']`

are both lists of length 3

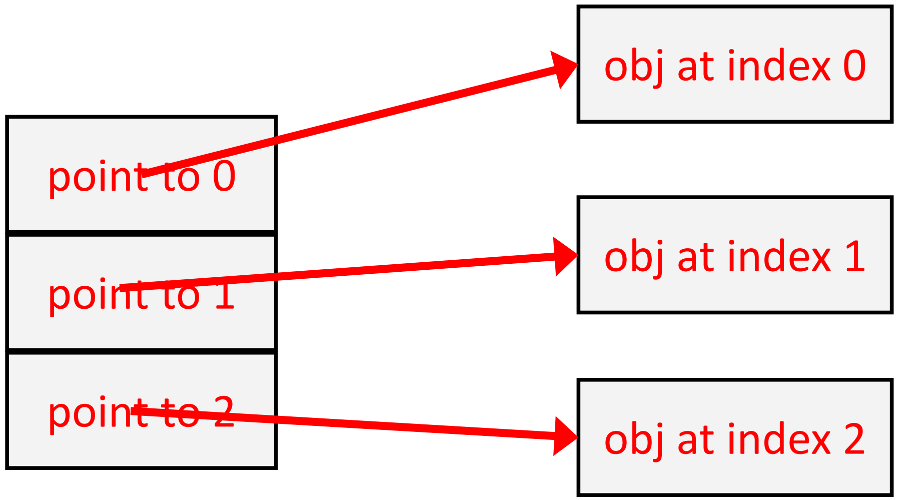
A common representation:

indirection!

a structure of 3 links/pointers to the object members

Python list: example

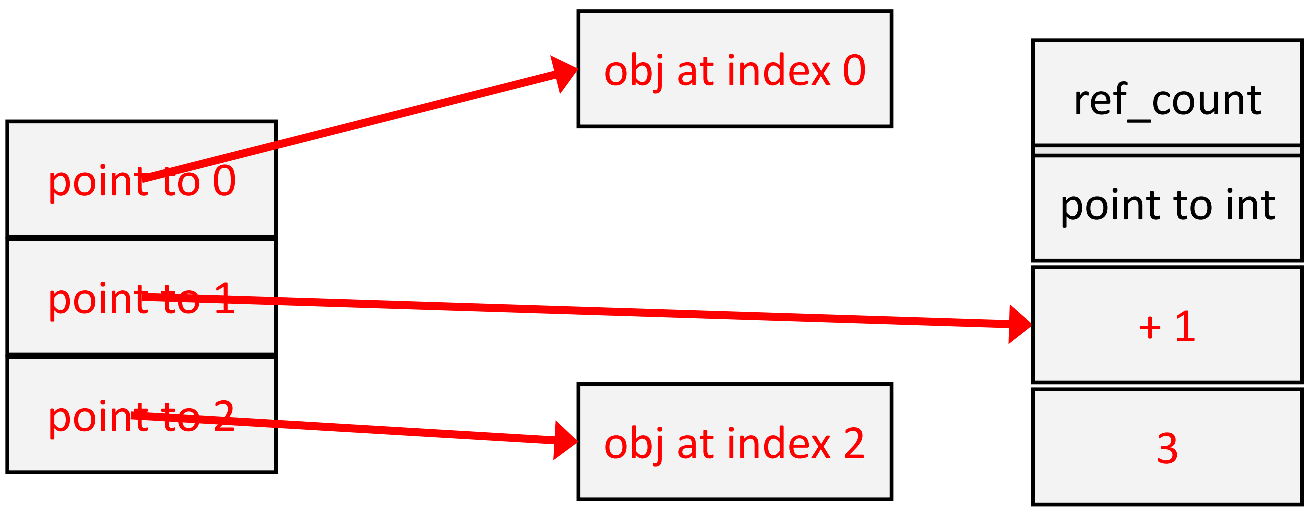
three element list



Python list: example

the three element list

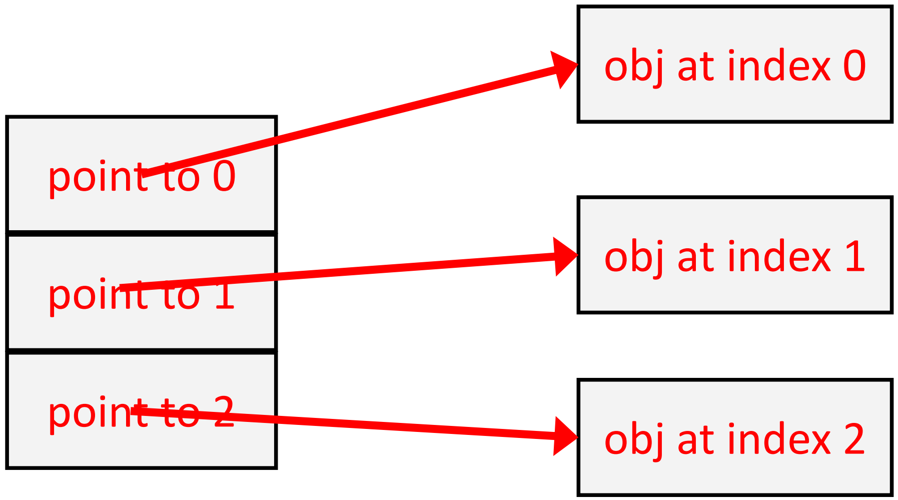
[ob0, **3**, ob2]



Python list: example

three element list:

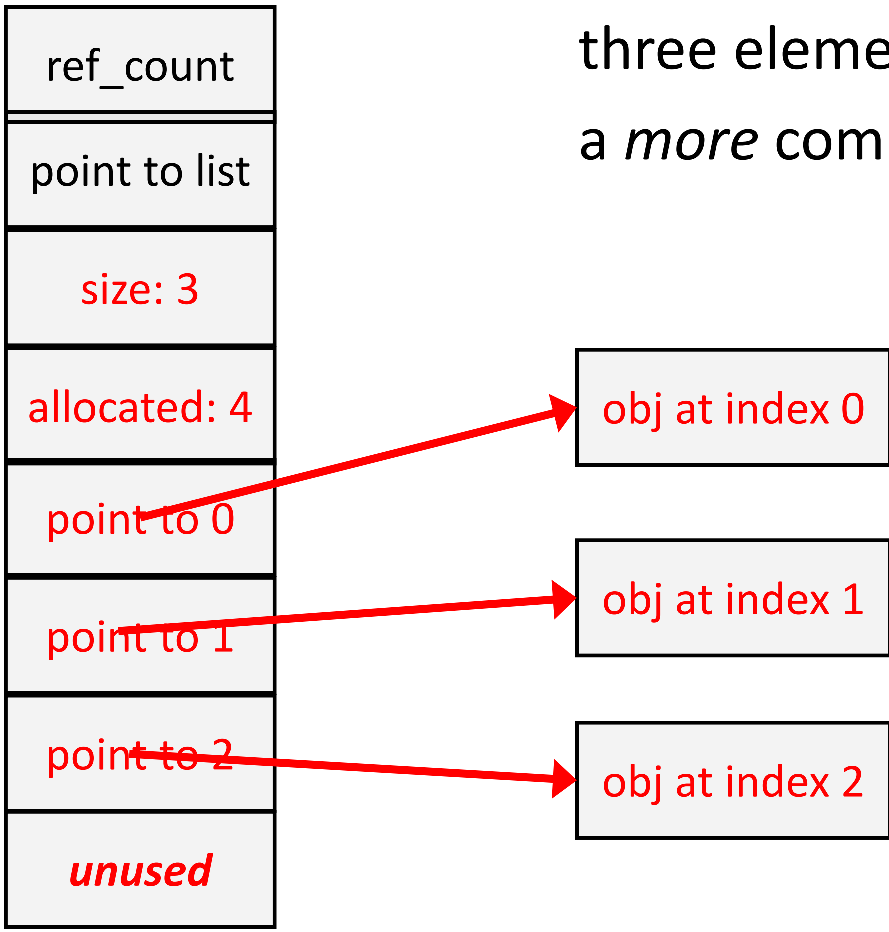
a *more* complete picture



Python list: example

three element list:

a *more* complete picture



Python list: the empty list

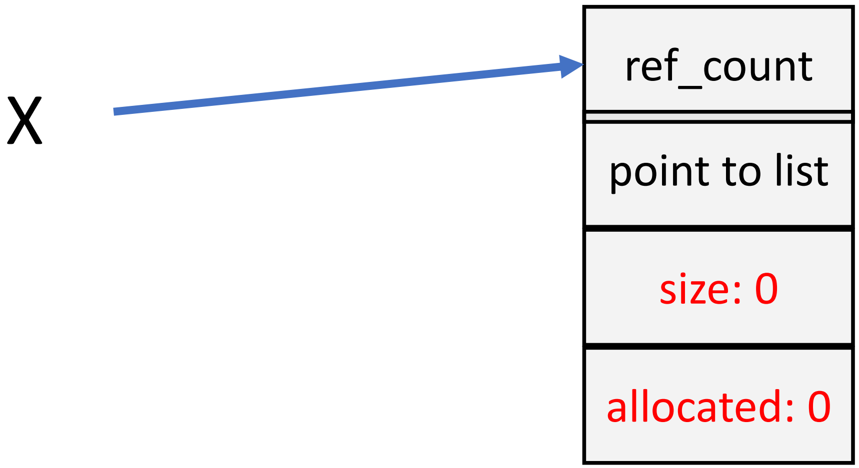
X=[]

From now on

we forget "ref count" and

"pointer to class"

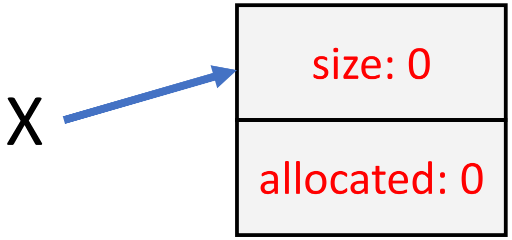
for graphical simplicity



constant-time op

Python list: appending on empty list

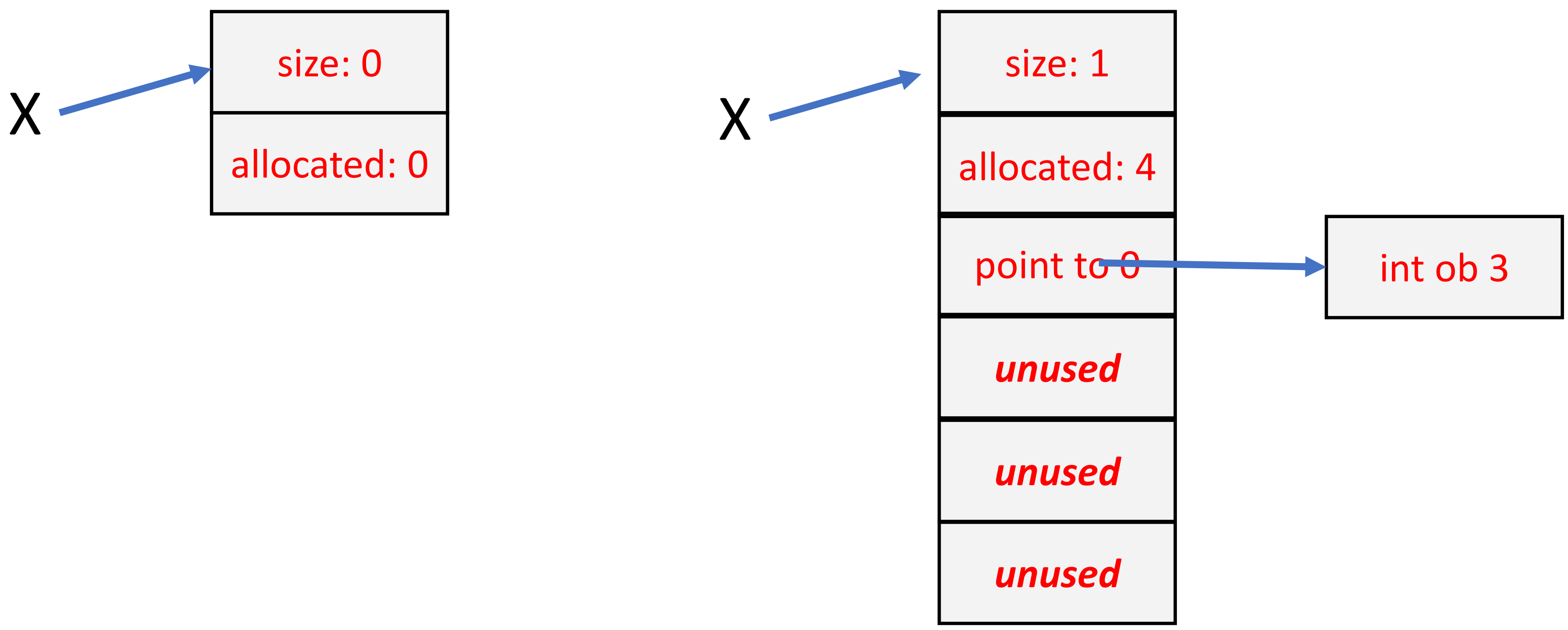
```
X=[]
X.append(3)
```



Python list: appending on empty list

X=[]

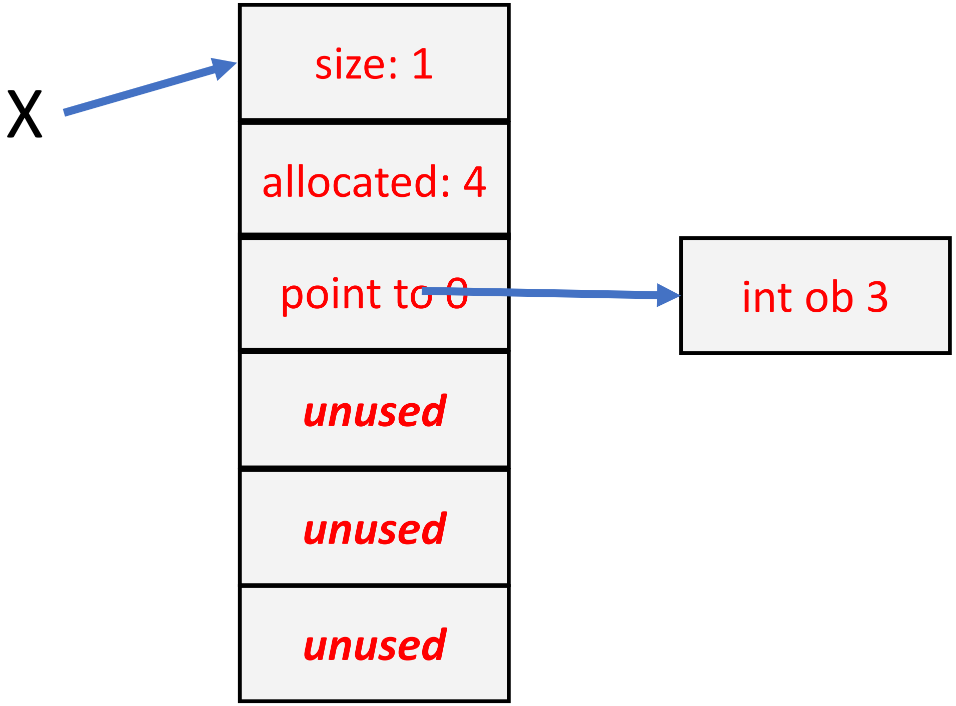
X.append(3)



Python list: appending on non empty list

X=[3]

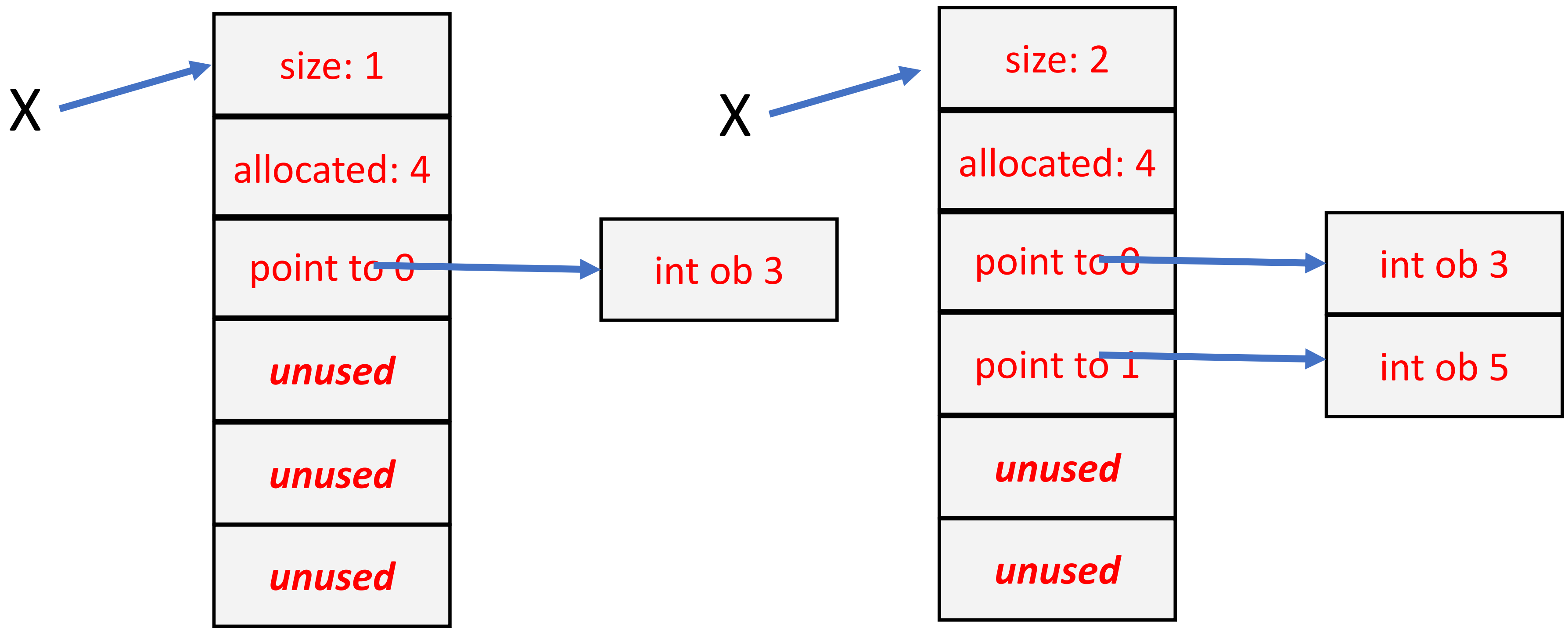
X.append(5)



Python list: appending on non empty list

X=[3]

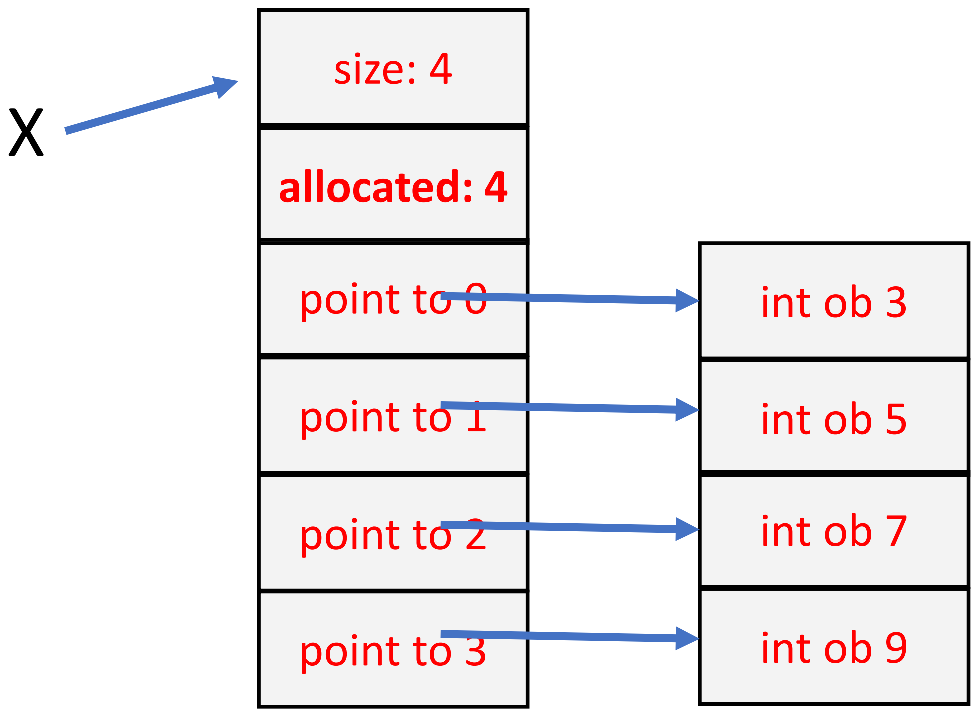
X.append(5)



Python list: appending on non empty list

X=[3,5,7,9]

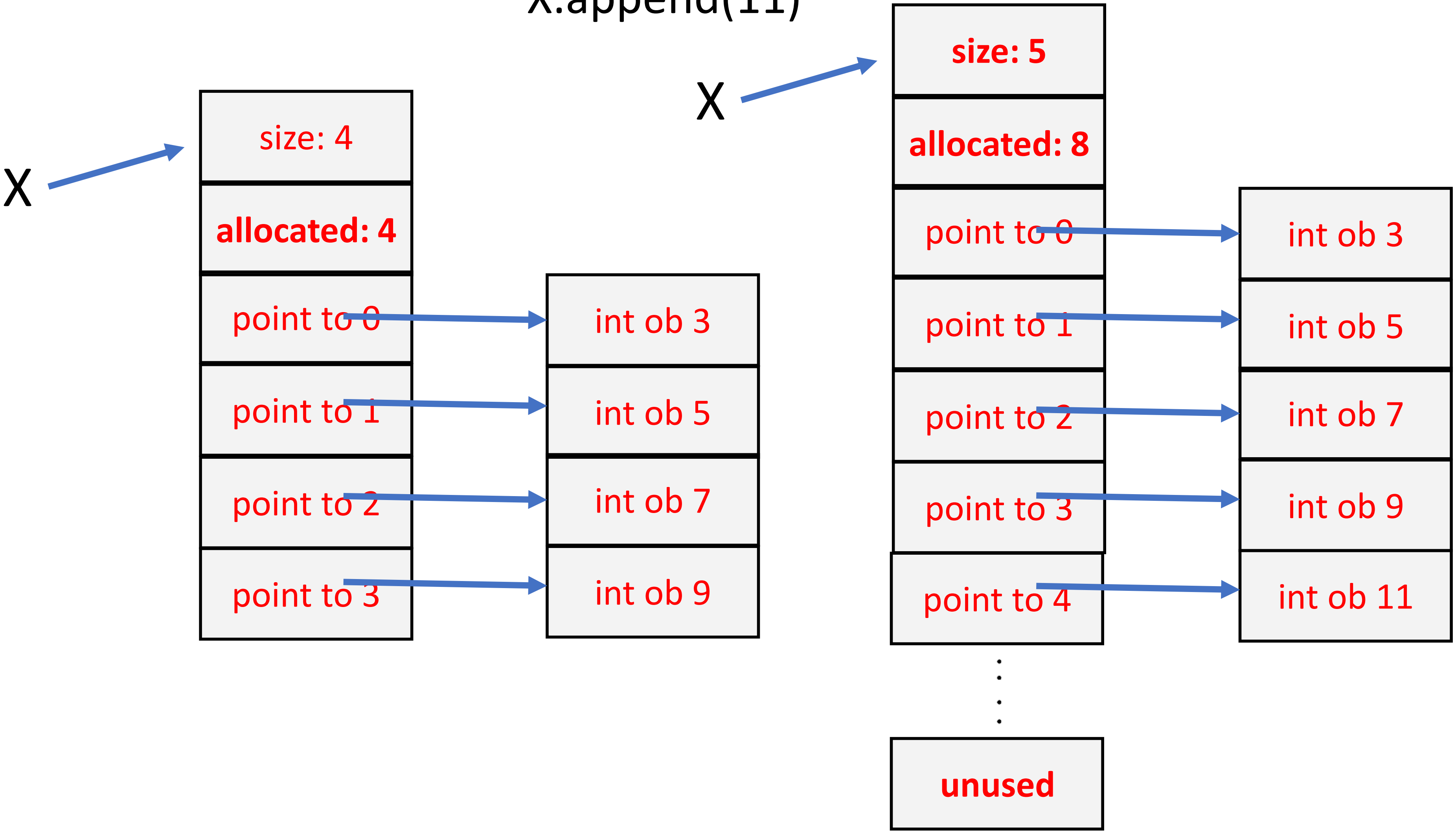
X.append(11)



Python list: appending on non empty list

X=[3,5,7,9]

X.append(11)



Python list: append

if there is an **unused** slot for the new object (pointer): used it

Otherwise, over-allocate a chunk of pointers; use the first

Over-allocation follows the pattern:

0, 4, 8, 16, 25, 35, 46, 58, 72, 88, …

Cost:

there are unused slots: constant-time

allocation is needed: time linear in the size

however *next* appends will be constant

In a long sequence of appends, the *amortized cost* of each one is constant

Python list: append

if there is an **unused** slot for the new object (pointer): used it

Otherwise, over-allocate a chunk of pointers; use the first

Over-allocation follows the pattern:

0, 4, 8, 16, 25, 35, 46, 58, 72, 88, …

Cost:

there are unused slots: constant-time

allocation is needed: time linear in the size

however *next* appends will be constant

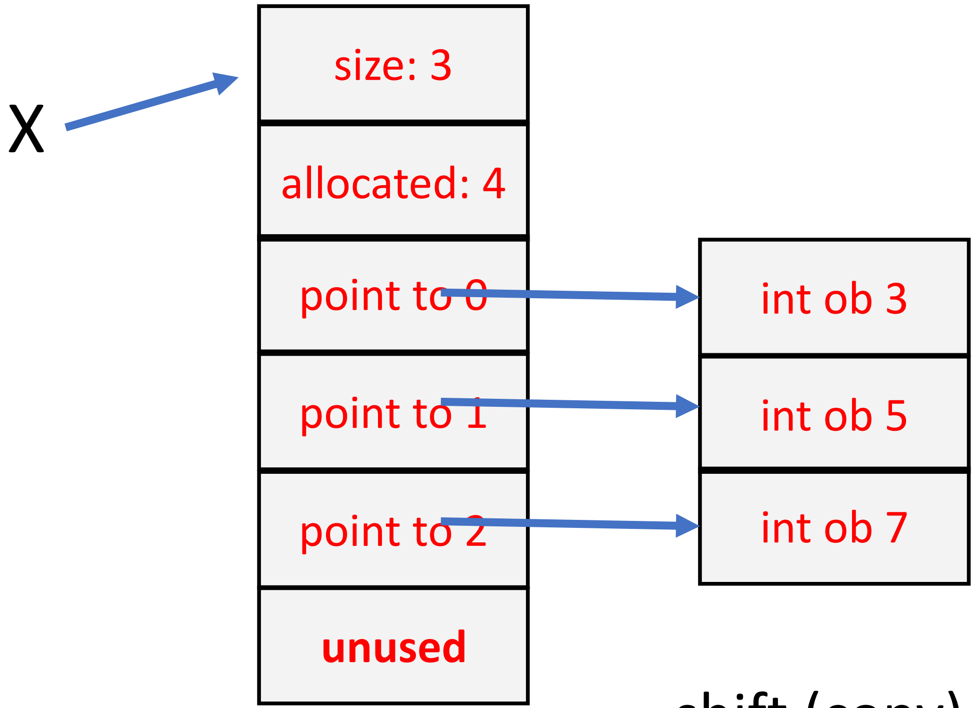
In a long sequence of appends, the *amortized cost* of each one is constant

constant-time op (amortized)

Python list: insert *inside* a non empty list

X=[3,5,7]

X.insert(1,9)



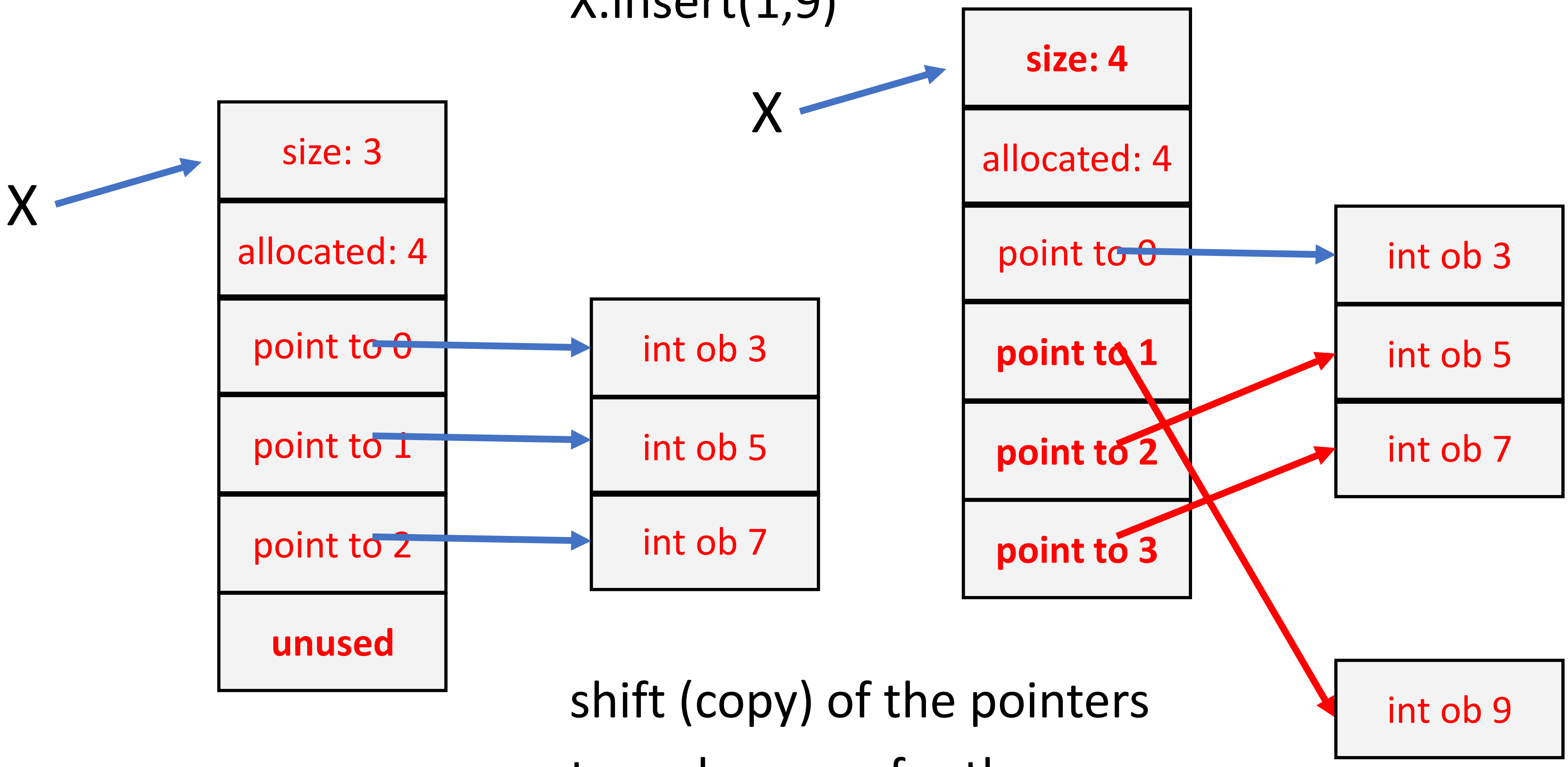
shift (copy) of the pointers

to make room for the new one

Python list: insert *inside* a non empty list

X=[3,5,7]

X.insert(1,9)



shift (copy) of the pointers

to make room for the new one

Python list: insert

if there is an **unused** slot for the new object (pointer):

shift the pointers, use the available one

Otherwise, **over-allocate** a chunk of pointers;

shift the pointers; use the available one

Cost:

there are unused slots: time linear in the len

allocation is needed: time linear in the len

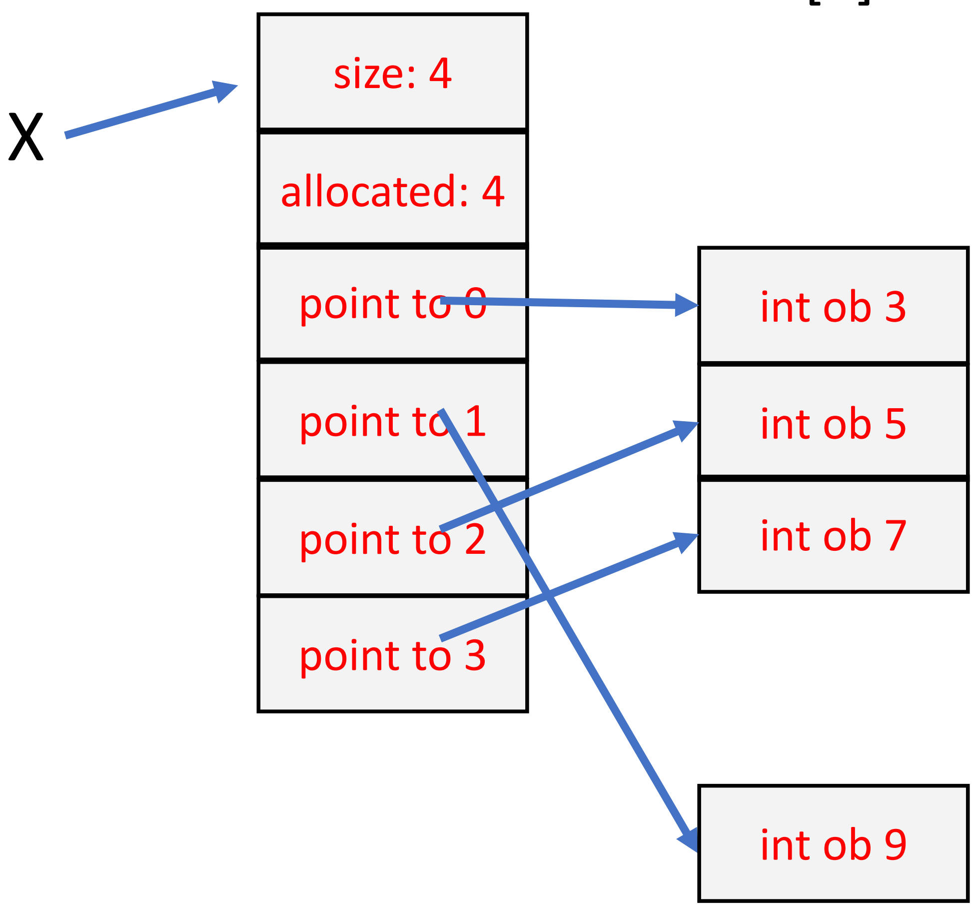
+ time for allocation (this will save time later)

linear time: O(n)

Python list: del

X=[3,9,5,7]

del L[0]

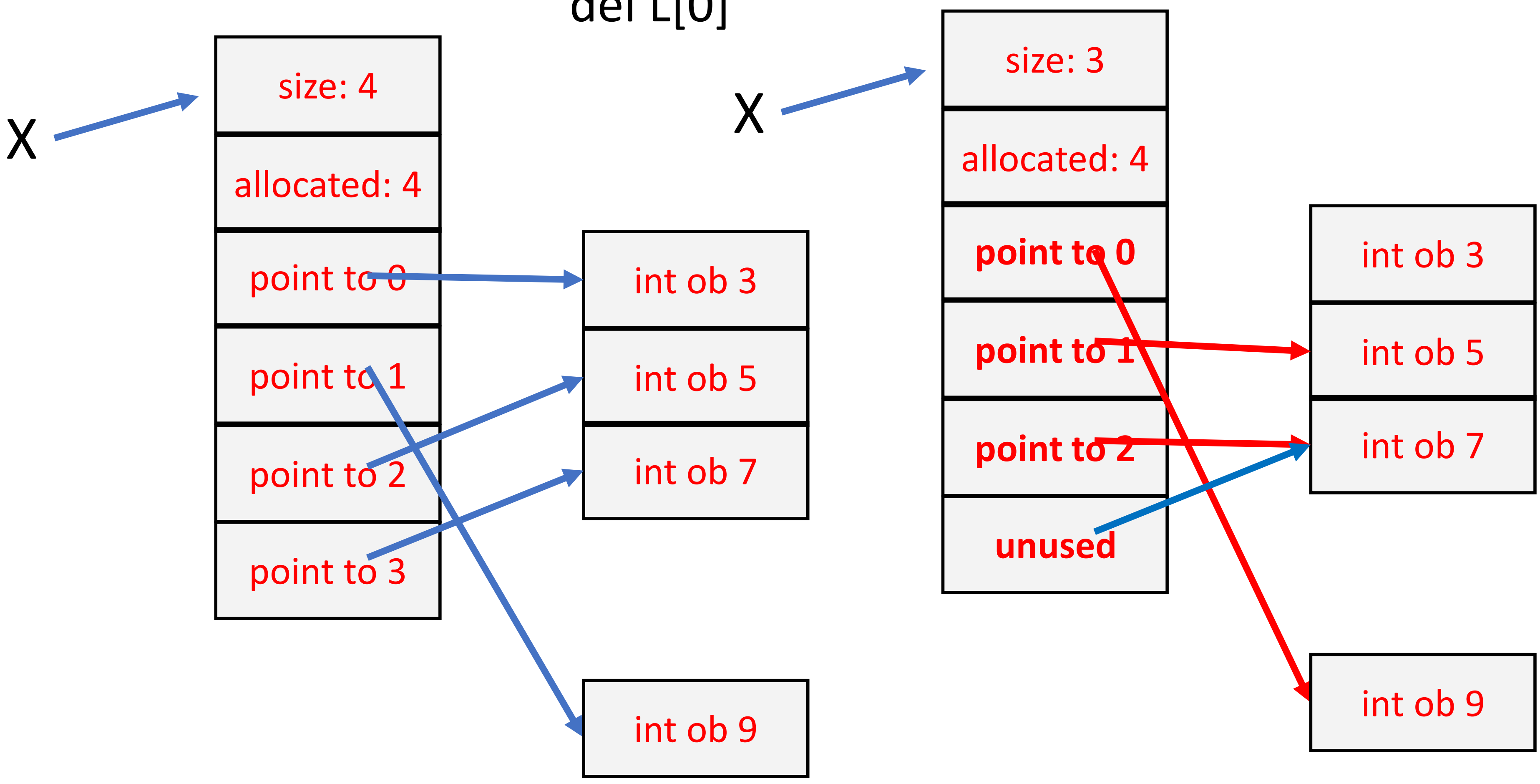


shift (copy) of the pointers

Python list: del

X=[3,9,5,7]

del L[0]



shift (copy) of the pointers

Python list: del, remove

Shift the pointers

Cost:

time linear in the len

When size goes below some threshold,
also deallocate a chunk of pointers

linear time: O(n)

Complexity of list operations

Operation	Complexity	Usage
List creation	O(n) or O(1)	x = list(y)
indexed get	O(1)	a = x[i]
indexed set	O(1)	x[i] = a
concatenate	O(n)	z = x + y
append	O(1)	x.append(a)
insert	O(n)	x.insert(i,e)
delete	O(n)	del x[i]
equality	O(n)	x == y
iterate	O(n)	for a in x:
length	O(1)	len(x)
membership	O(n)	a in x
sort	O(n log n)	x.sort()

Complexity

Remember: O(1) is constant-time

sum, max, min: O(n)

Operation	Complexity	Usage
List creation	O(n) or O(1)	x = list(y)
indexed get	O(1)	a = x[i]
indexed set	O(1)	x[i] = a
concatenate	O(n)	z = x + y
append	O(1)	x.append(a)
insert	O(n)	x.insert(i,e)
delete	O(n)	del x[i]
equality	O(n)	x == y
iterate	O(n)	for a in x:
length	O(1)	len(x)
membership	O(n)	a in x
sort	O(n log n)	x.sort()

Complexity

https://www.geeksforgeeks.org/complexity-cheat-sheet-for-python-operations/