Defining Classes in Python

Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

Classes in Python

As in C^{++} and Java, it is possible to define classes in Python.

There is some similarity, but also important differences.

Definition of a Class

The simplest possible class definition has form:

```
class Test :
   pass
```

The class is called `Test`. In Python, class names should start with a capital.

Class objects can be obtained by typing:

```
t1 = Test( )
t2 = Test( )
```

`t1` and `t2` have no fields.

No Static Typing!

Python has no static type system.

A class object has no fixed set of fields.

A field can be created by assigning to it.

Every field can be deleted by `del`.

Everyone can do that, because there are no private fields.

Because of this, different objects of the same class can have different fields.

No Static Typing! (2)

```
t1 = test.Test( )
t2 = test.Test( )

t1. greet = "good morning"
t2. greet          # undefined

t2. pi = 3.1415
t1. pi             # undefined

del t2. pi         # remove field pi of t2
```

Classes are Mutable Objects with Reference Semantics

```
t1 = test.Test( )
t2 = test.Test( )
t3 = t1

t1. greet = "good morning"
t3. greet    # prints good morning

del t3. greet
t1. greet    # is undefined
```

Initialization of a Class Object

One can define an initalization function `__init__( self )`.

If it exists, it is called when `Test( )` is called.

The `__init__` function can create and initialize fields. Fields are created simply by assigning to them:

```
class Test :
   def __init__( self ) :
      self. a = 1          #  Now the field is there.
      self. b = 2          #  Creates another field.
```

Access to the Class Object as `self`

In C^{++} and Java, member functions automatically have access to the fields of their class. In Python, one has to use `self`:

```
def update( self, a, b ) :
    self. a = a
    self. b = b
```

It can be called as `t1.update(100,200)`.

Equality is by Default Object Identity

This means that two objects are equal if they can be traced back to the same creation point.

If one compares two objects, Python will compare the addresses. If you need equality based on value, you will need to define `__eq__( )`.

If you call `t1 == t2`, Python will call `t1.__eq__( t2 )` if it exists. Otherwise, it compares addresses.

Since hashing has to consistent with `==`, you must also define `__hash__( )`. The interpreter will complain if you define `__eq__( )` without defining `__hash__( )`.

If you try to print `t1` and `t2`, it also prints the addresses. If you don't like that, you need to define `__repr__( )`.

Init Functions

In Python, an initialization function has form `__init__( self )`.

It creates and initializes the fields of `self`.

```
class Rational :
   def __init__( self, num, denom = 1 ) :
      self. num = num
      self. denom = denom


rat = Rational( 1, 2 )    # Constructs 1/2
rat = Rational( 5 )       # Constructs 5.
```

Difference with C^{++} : Member Initializers

C^{++} has member initializers in constructors:

```
rational( int num, int denom )
    : num( num ), denom( denom )
{ (constructor body) }
```

C^{++} needs them because it has value semantics. The space for the fields is created before the constructor code starts. Initialized objects can hold resources while unitialized memory cannot. This makes it necessary to distinguish between initialization and assignment. Since the code between `{ }` can only assign to the fields of `rational`, initialization has to be done before the first `{`.

Python does not need this distinction, because unitialized fields simply don't exist.

How is this problem solved in Java?

Difference with C^{++} : Copy Construction, Assignment, Destruction

Python doesn't have these things.

In C^{++}, the purpose of copy constructors is to preserve value semantics, to make sure that the object is really copied.

C^{++} : Assignment = destruction of old value + copy construction. (or copy construction + move)

The purpose of destructors is to return resources that were held by the object.

Python has reference semantics, and resources are returned by the garbage collector.

Printing

In order to be able to print a class object, define a method `__repr__( self )`.

`__repr__( )` must return a string. The string will be used when you print the object from the command line.

For example printing a rational number:

```
def __repr__( self ) :
   if self. denom == 1 :
      return '{}'. format( self.num )
   else :
      return '{} / {} '. format( self.num, self.denom )
```

Whenever a `Rational` is printed, it will be converted into a string by `__repr__( self )__`.

Pretty Printing

It is also possible to define a `__str__( self )` method, which will be preferred by `print( )`, when present.

The output of `__str__( self )` should be for the user. To be precise: Purpose of `__repr__( )` is to represent the object accurately, while `__str__( self )` should be readable for the user.

Arithmetic Operators

If you want to define an arithmetic operator, use:

`__neg__`	`-`	(unary minus)
`__add__`	`+`	(binary plus)
`__sub__`	`-`	(binary minus)
`__mul__`	`*`	(multiplication)
`__truediv__`	`/`	(real division)
`__floordiv__`	`//`	(integer division)
`__mod__`	`%`	(modulo)
`__pow__`	`**`	(power)
`__matmul__`	`@`	(matrix multiplication)

Overloaded Operators, Run Time Polymorphism

I said before that Python has no run time (or any other) overload resolution, but this is not quite true.

Python has run time overload resolution for class methods, when defined in different classes.

`t1.f( )` can be different from `t2.f( )` if `t1` and `t2` belong to different classes. The interpreter decides at run time.

There is no way to overload `t1.f( args )` for different types of `args`.

When evaluating `t1 + t2`, Python will look at the type of `t1`. If the type of `t1` is a class that has a definition of `__add__( )` defined, Python will call it.

This allows some overloading for built-in operations. For a fixed type of `t1`, you cannot overload `t2`.

If you want polymorphism in two arguments, you will have to implement it for the other argument by yourself:

```
def __add__( self, other ) :
   if not isinstance( other, Complex ) :
         other = Complex( other, 0 ) # change type.

   return Complex( self.re + other.re, self.im + self.re )
```

Polymorphic Operators (2)

Another example:

```
def __add__( self, other ) :
   if isinstance( other, str )
      raise NotImplementedError
         # Dont know how to add string to complex

   if not isinstance( other, Complex ) :
      return Complex( self.re + other, self. im )
   else :
      return Complex( self.re + other.re,
                      self.im + self.im )
```

Making sure that all cases are covered is hard.

Making all these decision at run time makes Python slow.

Reversed Operators

Polymorphic `__add( )__` can be used to add objects of any type to a class member: `c + 44`, `c + Complex(1,2)`, `c + "aaa"`

(if you are willing to write the `isinstance`s)

But we also want to evaluate expressions of form `1 + c`, where the second argument belongs to the class, but not the first.

For this, Python has reverse operators.

C^{++} does not have them, but should have them. Especially for `operator <<`, it would be useful.

Reverse Operators (2)

Reverse operators are tried by Python when no direct (not reversed) operator can be found, and the arguments have different types.

This means that `c1 + c2` is evaluated as follows:

1. First try to find `c1.__add__(c2)`. If this method exists, then it is called.
2. If `c1,c2` have the same type, then give up.
3. Otherwise, try to find `c2.__radd__(c1)`. If it exists, then call it.

Note that the arguments are exchanged in `__radd__( self, other )`, because the `self` argument must be always be first.

Reverse Operators (3)

`__radd__`	`+`	(binary plus)
`__rsub__`	`-`	(binary minus)
`__rmul__`	`*`	(multiplication)
`__rtruediv__`	`/`	(real division)
`__rfloordiv__`	`//`	(integer division)
`__rmod__`	`%`	(modulo)
`__rpow__`	`**`	(power)
`__rmatmul__`	`@`	(matrix multiplication)

In Place Operators

In place operators are operators of form

`c1 += c2,  c1 *= c2,` etc.

If you don't define them, Python evaluates them as

`c1 = c1 + c2,  c1 = c1 * c2`

In Place Operators (2)

`__iadd__`	`+=`	(binary plus)
`__isub__`	`-=`	(binary minus)
`__imul__`	`*=`	(multiplication)
`__itruediv__`	`/=`	(real division)
`__ifloordiv__`	`//=`	(integer division)
`__imod__`	`%=`	(modulo)
`__ipow__`	`**=`	(power)
`__imatmul__`	`@=`	(matrix multiplication)

In Place Operators (3)

Why does Python have in place assignment operators? Good Question, I don't know.

In C^{++}, the reason is clear, to avoid object copying, which is expensive.

In Python, there is no object copying, and nobody cares about efficiency anyway.

So the answer must be: You want to change a complete object, and you want all variables that refer to the same object, to change too.

In that case, you cannot use assignment, because that would replace the object in the variable, but not change the other copies.

In Place Operators (4)

In order to work well, in place assignment operators must return `self`.

A little bit of experimentation reveals that, otherwise, the behavior of the inplace operators in Python is bizar.

When `+=` returns `self`:

```
c1 = Complex(1,2)
c2 = c1
d = Complex(10,11)
c1 += d
c1  # Meaningful behavior.
c2
```

In Place Operators (5)

Now try the same when the in place operator does not return a value:

```
c1 = Complex(1,2)
c2 = c1
d = Complex(10,11)
c1 += d
c1  # Prints nothing, because no return value.
c2  # Still modified.
```

So, we conclude that an in place operator first modifies its `self` argument (and all shared instances), and after that, assigns the return value to it.

Why this second, apparently redundant assignment?

There probably is a reason.

Comparison Operators, Hashing

If you define a class, Python by default defines equality as object equality.

```
Complex(1,2) == Complex(1,2)
   False

A = { }
A[ Complex(1,2) ] = 100
A[ Complex(1,2) ]
   # Error that Key is not found.
```

Comparison Operators

Python allows definition of the following comparison operators:

`__eq__`	`==`
`__ne__`	`!=`
`__lt__`	`<`
`__gt__`	`>`
`__le__`	`<=`
`__ge__`	`>=`

(Note that `<,>,<=,>=` have no meaningful definition on complex numbers.)

Comparison Operators (2)

Comparison operators can be made polymorphic in the same way as arithmetic operators.

There are no reverse operators, but comparison is automatically tried the other way:

```
c == d
c != d
   # First look in type of c.
   # If not found, look in type of d.
c < d
c > d
c <= d
c >= d
   # First look in type of c.
   # If not found, look for reverse in type of d.
```

Hash Functions

If you want to use your class as key in a dictionary, you have to define (in addition to `__eq__`) a hash function.

The best way to do this, is not to be creative by yourself, but to use hashing on tuples:

```
def __hash__( self ) :
    return hash( ( self. re, self. im ) )
```

Now `Complex` can be used with dictionary.

Application Operator

If you want to apply a class object as a function, use `__call__( self, v )` .

For example:

```
def __call__( self, v ) :
   return Vector( self. a11 * v.x + self. a12 * v.y,
                  self. a21 * v.x + self. a22 * v.y )
```

Matrix is an object, that can be applied on a vector.

Function Overloading

Overloading means that the same name can be used with different definitions. If the language has static type checking, the correct overload can be selected at compile time.

For example, a function `f` can have definitions
`f( int ),  f( double ),` etc.

If we know at compile time that `i` is a **int**, then we call the first version in `f(i)`.

C^{++} and Java have static type checking.

Function Overloading (2)

Python has overloading for class members. If a function `f` occurs as member in different classes, then `f` can be overloaded in expressions of form `t.f( )`.

Each class in Python can have at most one member function with a given name.

Function Overloading (3)

Apart from member functions, Python has no overloading.

If a function occurs with different definitions, the last definition is remembered, and earlier definitions are forgotten.

This also applies to the number of arguments, even though that would be easy to check statically.

Dynamic Function Overloading

If you want to use some function on different types, or different numbers of arguments, you will have to implement it by yourself:

Overloading `f` for different types:

```
def f( x ) :
   if isinstance( x, int ) :
      (do something with x as int)
   if isinstance( x, double ) :
      (do something with x as double)
```

Dynamic Function Overloading (2)

Overloading `f` for different numbers of arguments:

```
def f( x1, x2 = None, x3 = None ) :
   if x2 == None :
      (there is one argument)

   if isinstance( x2, int ) and x3 == None :
      (there are two arguments, x2 is an int.)
```

Dynamic Function Overloading (3)

There are two problems with dynamic overloading in code:

1. It is slow at run time.
2. One programmer must consider all the types that `f` can be called with, and has to deal with them.
3. There is no protection against missing types, against inconsistent definitions, etc.

Dangers of Default Parameters

Python allows default values for parameters, but these can be dangerous, because of reference semantics.

```
class A :
   def __init__( self, lst = [] ) :
      self. lst = lst


a1 = A( )
a2 = A( )
a1. lst. append(1)
a2. lst                    # also changed !
```

Dangers of Default Parameters (2)

It is better to write:

```
class A :
   def __init__( self, lst = None ) :
      if lst == None:
         lst = []
      self. lst = lst


   a1 = A( )
   a2 = A( )
   a1. lst. append(1)
   a2. lst                          # Still [].
```

Inheritance in Python

A class can be defined as a subclass of another by using `( )` in the class definition, as for example in `class Rational( Number ):`

Inheritance in Python is not as impressive as in other languages like for example C^{++} or Java, because the fields of an object do not depend on its type.

So what do we get?

- If class `A` is a subclass of `B` and `isinstance( t, A )` is true, then `isinstance( t, B )` is also true.
- In a call of form `t.f(  )`, if `t` has type `A`, and `class A` has no method `f` defined, then the interpreter will look in class `B` for an `f` method.

Inheritance in Python (2)

Everything ultimately inherits from `Object` in Python.

A class can inherit from more than one class. This is called multiple inheritance. In that case, member functions are looked up in depth-first, left-right order.

In C^{++}, multiple inheritance is quite tricky, because the class of an object determines which fields it has (and how they are laid out in memory). In Python, this problem doesn't exist, because there is no relation between type and fields.

As a general rule, one should avoid big hierarchies.

OO is in general overused. It has only a few, real applications: Graphical objects, file hierarchies.

Calling the Parent Class

If you want to call methods of a parent class, you can do this in two ways:

In case of single inheritance, you can call `super( ).f( )`, where `f` is the method in the parent class that you want to call. You don't need to know the name of the parent class.

In case of multiple inheritance (or you want to call the class by its name), use `name.f(self)`. In this case you need to include `self` as argument.

This syntax can also be used for special methods, like `__init( )__` or `__repr__( )`.

Simulating RAII in Python

C^{++} has destructors.

Although traditionally called 'destructor', there is no meaningful notion of destruction inside a computer. The task of the destructor is to return the resources that were held by the object.

The semantics of C^{++} guarantees that the destructor of a variable is always called when the variable goes out of scope. This is independent of how the variable's block is exited (`return`, `break`, `throw`, or by peacefully reaching the end.)

The idea that resources should always returned by destructors is covered under the name RAII (Resource Acquisition is Initialization)

Simulating RAII in Python (2)

In Python you can write:

```
f = open( name, "r" )

(read a little from file f)

f. close( )
```

If the code in the middle accidentally returns early, or raises an exception, `f.close( )` will not be executed.

Python has garbage collection, and the GC will eventually close the file, but it is not guaranteed when this happens.

Python has two ways of guaranteeing clean up: finally and the with-statement.

`with`-statement

```
with open( name, "r" ) as f :
   (do the reading from f)
```

File `f` is guaranteed to be closed, as soon as the block is exited, independent on how it is exited.

`with` works on any object that has `__enter__( )` and `__exit__( )` methods. It does the following:

1. Call `__enter__( )`, and store the result in the variable.
2. Calls `__exit__( )` when the block is left.

For example, the file type in Python has `__enter__()` and `__exit__( )` methods.

The `__exit__( )` method must have four arguments:

```
__exit__( self, extype, exvalue, traceback )
```

`extype` is the type of the exception, `exvalue` is the exception itself, `traceback` is the traceback from the point where the exception was raised.

In case the `do` block exited normally, all three parameters except `self` will be `None`.

The `__enter__( )` method has one argument, the class object. I think in most cases, it does nothing.

Finally

Another way of specifying code that must be executed in all cases, is a `finally` clause:

```
try:
    (risky code)
except:
    (catch some exceptions)
finally:
    (do necessary clean up)
```

The `finally` clause is guaranteed to be executed, independent of whether the risky code raises an exception.

Creating an Iterator from An Object

We have already seen one way of obtaining iterators, namely by a co-routine that `yield`s instead of `return`s.

Iterators can also be constructed from objects.

Python distinguishes between iterables and iterators. The iterable creates the iterator. In a **for** loop, list constructor or generator expression, the user has to provide an iterable.

From an iteratable, an iterator can be constructed by calling its `__iter__( self )` method.

The iterator must have a method `__next__( self )__`, which keeps on returning a next object, until it raises `StopIteration( )`.

Why does Python distinguish iterator from iterable?

Because the same iterable can simultaneously occur in different for loops (or other contexts using an iterable.)

For example:

```
it = ...
for i in it :
   for j in it :
      print( i, j )
```

If both **for**-loops would work directly on `it` itself, state would be shared, and behaviour would be unexpected.

The **for** loops extract iterators (by calling `iter( it )` resulting in two independent iterators, and things will work as expected.

Range as Class, not Coroutine

```
class Rrrange :
   def __init__( self, start, end ) :
      self. start = start
      self. end = end

   def __iter__ ( self ) :
      return Iiiter( self. start, self. end )
```

```
class Iiiter :
   def __init__( self, start, end ) :
      self. start = start
      self. end = end
      self. counter = start

   def __next__ ( self ) :
      if self. counter >= self. end :
         raise StopIteration( )
      val = self. counter
      self. counter += 1
      return val
```

One can write:

```
r = Rrrange(1,4)
for x in r :
   for y in r :
      print( x, y )
```

Type Hints (MyPy)

Python has no static typing, but it has something called type hints.

These type hints are completely optional. They are ignored by the interpreter. A separate program, called MyPy can be used for checking types.

If MyPy complains, your program might still work.

I find MyPy very useful, and I recommend that you always use it.

Static type checking is especially useful for managing changes.

Type Hints (2)

Possible types are:

```
List[ X ]      #  List of X
Tuple[ X1, ...,  Xn ]
   # Tuple (fixed size) of X1, ... Xn
Optional[ X ] # An X or None
Union[ X1, ..., Xn ]   # An X1, X2, ..., or Xn.
Dict[ X, Y ]  # A dictionary with keys X and values Y
Set[ X ]                # Set of X
int, str, float, bool  # primitive types
```

In addition to this, every defined class can be used as type.

You need to add `from typing import X`, for the non-primitive types that you want to use.

Type Hints (3)

In order to hint the type of variables, write `field : Type` directly after the class definition, e.g.

```
class Test :
   a : int
   b : int
```

Functions can be typed as follows:

```
def gcd( i1 : int, i2 : int ) -> int :
```

Use `None` when the function returns no value.

Iterators can be typed with

```
def fib( n : int ) -> Iterator[ int ] :
```

Variables can be assigned with a type:

```
hello : str = "hello world"
```

Type Hints (4)

In order to cast an expression `e` to a more specific type (needed when more becomes known about an `Optional` or `Union`, use `cast( Type, e )`. In order to use `cast`, add
`from typing import cast`

See `https://mypy-lang.org/` for documentation of MyPy.

Type Hints (5)

```
from typing import Optional
from typing import cast
u1 : Optional[ int ]
u2 : int

if u1 != None :
   u2 = u1  # MyPy doesn't like it.

if isinstance( u1, int ) :
   u2 = u1  # MyPy accepts it

if u1 != None :
   u2 = cast( int, u1 )
      # MyPy accepts it. At run time, the cast
      # does nothing.
```