Classes 3

Thomas Schwarz, SJ

Repetition

- Functions are usually defined with def

```
def quad_form(a, b, c, x, y):
    return a*x**2+b*x*y+c*y**2
```

- We specify parameters by:
  - Position

```
quad_form(1,3,2,1,1)
```

  - Keyword

```
quad_form(b=3,a=1, c=2, x=1, y=1)
```

Repetition

- Default arguments:
  - We can give default arguments
    - Python syntax restricts us to have default arguments after positional arguments
  - Work well with keyword arguments

```
def quad_form1(a=1, b=1, c=1, x=0, y=0):
    return a*x**2+b*x*y+c*y**2

quad_form1(x=2, y=3)
```

Repetition

- Keyword arguments:
  - A necessity if there are many parameters
    - Which is the case for most library functions in numpy, scipy, pyplot, …
- Default arguments:
  - Consider using them as much as possible

Repetition

- Anonymous functions
  - Allow you to define a function on the fly

```
integrate(lambda x,y: x**2+x*y,
          0,
          1,
          3,
          4)
```

  - Keyword is lambda, followed by arguments followed by colon followed by an expression

Repetition

- Ternary expression
  - Boolean condition that determines the value
  - After new value when true
  - Before new value when false

```
min = a if a < b else b
```

  - Replaces the multiline if-else and can be used within a lambda function

Classes through Special Methods

- Python:
  - Many mechanisms use specialized (= dunder) methods

Classes through Special Methods

- Example: Playing cards (again)

```
class Card:
    def __init__(self, suite, rank):
        self.suite = suite
        self.rank = rank
    def __str__(self):
        return "({:2s},{:2s})".format(
                     self.suite[:2],
                     self.rank[:2])
    def __repr__(self):
        return '[Card' + str(self)+']'
```

Classes through Special Methods

- Can find all attributes of an instance defined using __dict__ or dir :

```
>>> c=Card('heart', 'king')
>>> c.__dict__
{'suite': 'heart', 'rank': 'king'}
```

```
>>> dir(c)
['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__retr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'rank',
'suite']
```

Classes through Special Methods

- Equality versus Identity
  - Default evaluation for `==` looks at location of storage
    - Can get storage location with object.__repr__( )
    - Or in most Python implementation, with id

```
>>> id(d)
140299613922544
>>> object.__repr__(d)
'<__main__.Card object at 0x7f9a0ca664f0>'
>>> hex(id(d))
'0x7f9a0ca664f0'
```

Classes through Special Methods

- Equality versus Identity
  - This is usually not the behavior we want
    - Equality means all attributes are equal
    - Need to define __eq__ in your class

```
class Card:
    def __eq__(self, other):
        return self.suite==other.suite and self.rank==other.rank
```

```
>>> d=Card('heart', 'king')
>>> c=Card('heart', 'king')
>>> d==c
True
```

Classes through Special Methods

- Equality versus Identity
  - We can still compare for identity with **is**

```
>>> d is c
False
```

Classes through special methods

- Identity, equality, equality of names are all different concepts
  - As the following excerpt will show

Classes through Special Methods

'You are sad,' the Knight said in an anxious tone: 'let me sing you a song to comfort you.'
'Is it very long?' Alice asked, for she had heard a good deal of poetry that day.
'It's long,' said the Knight, 'but very, *very* beautiful. Everybody that hears me sing it—either it brings the *tears* into their eyes, or else—'
'Or else what?' said Alice, for the Knight had made a sudden pause.
'Or else it doesn't, you know. The name of the song is called "*Haddocks' Eyes*."'
'Oh, that's the name of the song, is it?' Alice said, trying to feel interested.
'No, you don't understand,' the Knight said, looking a little vexed. 'That's what the name is *called*. The name really *is* "*The Aged Aged Man*."'
'Then I ought to have said "That's what the *song* is called"?' Alice corrected herself.
'No, you oughtn't: that's quite another thing! The *song* is called "*Ways and Means*": but that's only what it's *called*, you know!'
'Well, what *is* the song, then?' said Alice, who was by this time completely bewildered.
'I was coming to that,' the Knight said. 'The song really *is* "*A-sitting On A Gate*": and the tune's my own invention.'

Classes through special methods

- See:
    - Name of the name
    - Name
    - Call
    - Identity

Classes through Special Methods

- We cannot make cards into elements of sets without making them hashable

```
>>> seta = {c}
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    seta = {c}
TypeError: unhashable type: 'Card'
```

-

Classes through Special Methods

- Need to declare a method __hash__ and a method __eq__
    - ```
      class Card:
          def __hash__(self):
              return hash(self.suite)*hash(self.rank)
      ```
    - Now it works

```
>>> c = Card('heart', 'king')
>>> seta = {c}
>>> c in seta
True
```
```

Classes through Special Methods

- But to do this, we should make cards immutable
  - Right now, we can just say

```
c.rank = 'ace'
```

- Strategy: declare the components private
- Create a getter function
  - Which we do by using a property generator

Classes through Special Methods

- Implementation

```
class Card:
    def __init__(self, suite, rank):
        self._suite = suite
        self._rank = rank
    @property
    def suite(self):
        return self._suite
    @property
    def rank(self):
        return self._rank
```

private attributes

Classes through Special Methods

- Implementation

```
class Card:
    def __init__(self, suite, rank):
        self._suite = suite
        self._rank = rank
    @property
    def suite(self):
        return self._suite
    @property
    def rank(self):
        return self._rank
```

made to behave like attributes

Classes through Special Methods

"Perl does not have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shot-gun."

- -Larry Wall, Creator of Perl

Classes through Special Methods

- Containers:
  - Example: a deck of cards

```
class Deck:
    def __init__(self, suites, ranks):
        self.cards = [Card(s,r) for s in suites for r in ranks]
    def __str__(self):
        retVal = [ ]
        for card in self.cards:
            retVal.append(str(card))
        return '\n'.join(retVal)
```

Classes through Special Methods

- We want:
  - Sequences: length and [ ]
  - Slicing
  -

Classes through Special Methods

- Implementing sequencing
  - Define __len__ and __getitem__

```
class Deck:
    def __len__(self):
        return len(self.cards)
    def __getitem__(self, i):
        return self.cards[i]
```

Classes through Special Methods

- Now we can do the following:
  - Get an element
  - Randomly select
  - Use slices

```
>>> import random
>>> deck = Deck(suites, rank)
>>> random.choice(deck)
>>> print(deck[5:10])
>>> print(deck[3])
```

Classes through Special Methods

- But we cannot shuffle a deck of cards

```
>>> random.shuffle(deck)
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    random.shuffle(deck)
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/
python3.8/random.py", line 307, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'Deck' object does not support item assignment
```

Classes through Special Methods

- We need to implement a __setitem__ method

```
def __setitem__(self, position, card):
        self.cards[position] = card
```

Classes through Special Methods

```
>>> deck = Deck(suites, ranks)
>>> import random
>>> random.shuffle(deck)
>>> print(deck)
(cl,ki)
(di,ja)
(cl,4 )
(he,3 )
(cl,9 )
```

Classes through Special Methods

- We could even use **monkey-patching**
  - Define a function that takes deck, position, and card as arguments
  - Dynamically create a Deck.__setitem__ method

```
Deck.__setitem__ = setcard
```

Inheritance

"We started to push on the inheritance idea as a way to let novices build on frameworks that could only be assigned by experts"

--Alan Kay: The Early History of Smalltalk

Inheritance

- To inherit from a class, just add the name of the base class in parenthesis

```
class BlackjackCard(Card):
```

Inheritance

- To initialize a derived class, usually want to call the initializer of the base class

```
values = {'ace':11, '2':2, '3':3, '4':4, '5':5, '6':6, '7':7, '8':8,
          '9':9, '10':10, 'jack':10, 'queen':10, 'king':10}

class BlackjackCard(Card):
    def __init__(self, suite, rank):
        super().__init__(suite, rank)
        self.value = values[rank]
        self.softvalue = 1 if rank=='ace' else self.value
    def __str__(self):
        return "{} of {} with value {}({})".format(
            self.rank,
            self.suite,
            self.value,
            self.softvalue
            )
```

Inheritance

- Notice:
  - All methods in the base class are still available and attributes
  - But we can also override them

```
def __hash__(self):
        return super().__hash__()^self.softvalue
```

Calling base class function

Inheritance

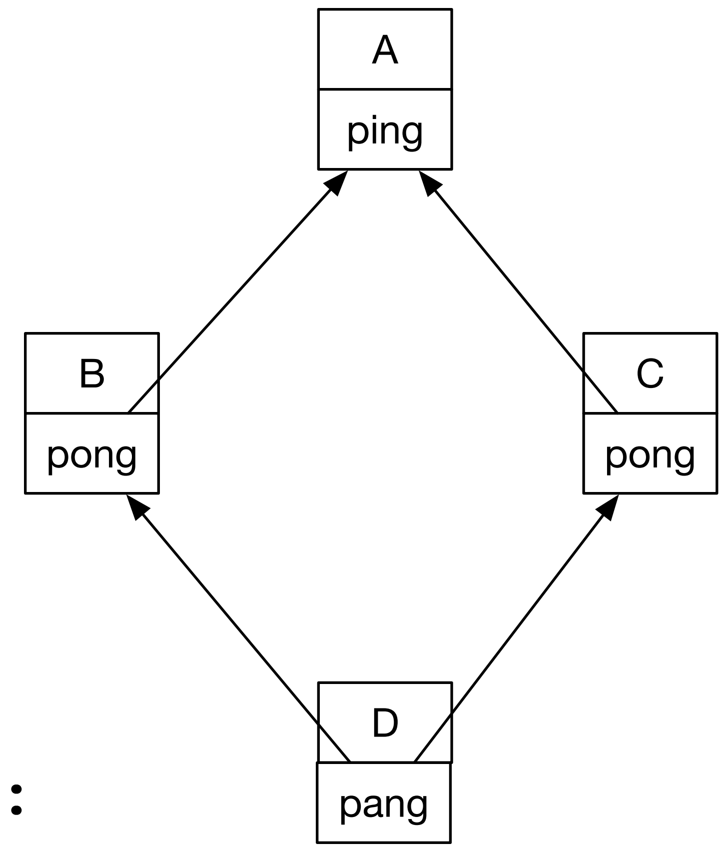
- Multiple inheritance
  - Allowed but tricky
  - **Diamond Problem**

```
class A:
   def ping(self):
      print('ping')

class B:
   def pong(self):
      print('pong')

class C:
   def pong(self):
      print('PONG')
```

```
class D(B,C):
   def ping(self):
      super().ping()
   def pang(self):
      super().ping()
      super().pong()
      C.pong(self)
```



Inheritance

- Method Resolution for d.pong( ):
  - First look in the current class
  - Then look into B
  - Then look into C
  - Then look into A
- Implemented via __mro__, which lists the classes in a certain order
- Can avoid ambiguity by giving explicit class names in the invocation

```
class D(B,C):
    def ping(self):
        super().ping()
    def pang(self):
        super().ping()
        super().pong()
        C.pong(self)
```

Inheritance

- Multiple inheritance can be used
  - Can use inheritance to define an interface:
    - A base class that requires that certain methods are implemented
  - Then multiple inheritance is fine

Operator Overloading

- Fundamental Rule:
  - Do not overload operators that do not make sense
    - E.g. Addition for cards makes no sense
    - Addition for complex numbers makes sense

Operator Overloading

- Unary Operations:
  - -   __neg__
    - Negative
  - + __pos__
    - +x is not always the same as x
  - ~ __inv__
    - Bitwise inverse of an integer

Operator Overloading

- Binary Operations
  - When confronted with an expression
    - `a ^ b`
  - Python looks into the class of a for a method `__xor__(self, other)`
  - If not found, then Python looks into the class of b for a method `__rxor__(self, other)`

Operator Overloading

- Binary Operations
  - When Python sees `a ^= b`
  - Then Python looks into the class of a for a method `__ixor__(a,b)`
    - `a = ixor(a,b)` is equivalent to `a^=b`

Operator Overloading

- Implementation:
  - All methods need to return an object
  - Operands do not have to be from the same class

Operator Overloading

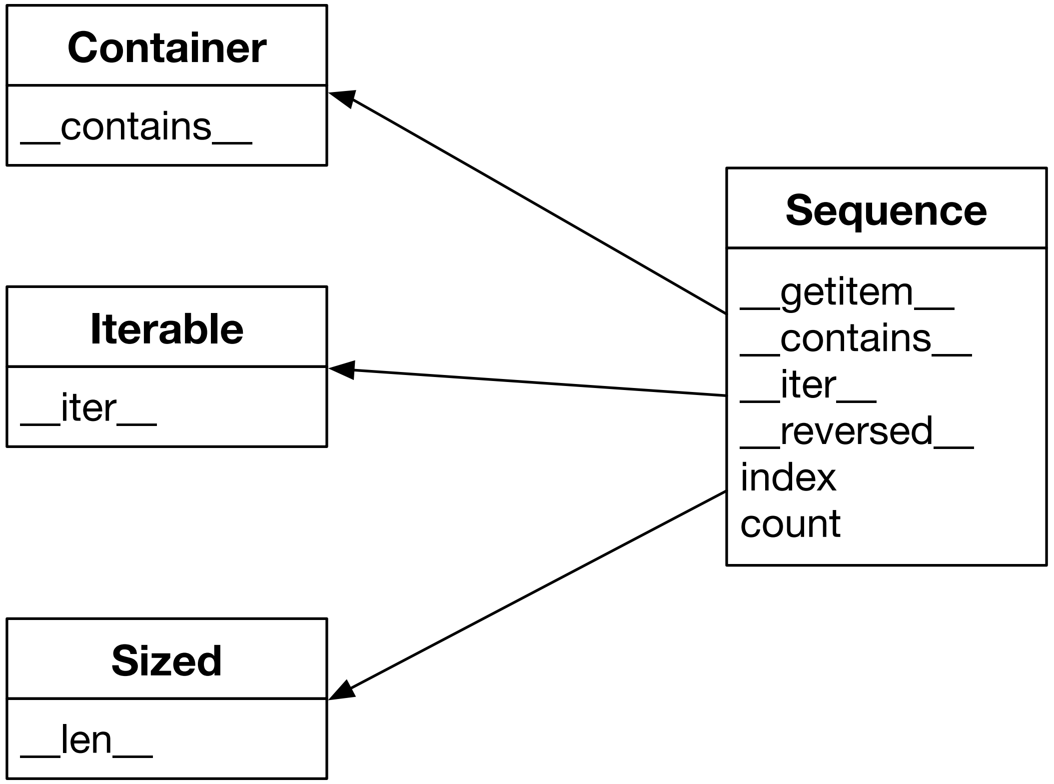
```
class Complex:
    def __init__(self, re, im):
        self.re = re
        self.im = im
    def __str__(self):
        return "({},{})".format(self.re, self.im)
    def __add__(self, other):
        return Complex(self.re+other.re, self.im+other.im)
    def __iadd__(self, other):
        self.re += other.re
        self.im += other.im
        return self
    def __radd__(self, other):
        return self+other
```

Interfaces

- Interfaces encapsulate how a user can use a certain set of classes
- Python does not need interfaces and only implemented them as Abstract Base Classes (ABC) in 3.4

Interfaces

- Example: Sequences



- An interface describes what can be invoked

Interfaces

- Example: Sequences
  - Some missing methods can be implemented via other methods
    - in still works even without __contains__ and __iter__

Interfaces

- ABC: Abstract Base Class
    - A class that does not have any methods implemented
- If you derive a class from an ABC:
    - You have to implement these methods
    - You make a public declaration that these methods are in your class

Interfaces

```
class FrenchDeck(collections.MutableSequence):
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                       for rank in self.ranks]

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

    def __getitem__(self, position):
        return self._cards[position]

    def __setitem__(self, position, value):
        self._cards[position] = value

    def __delitem__(self, position):
        del self._cards[position]

    def insert(self, position, value):
        self._cards.insert(position, value)
```

Interfaces

- Here we have to implement methods that do not make sense for a deck of cards because MutableSequence demands them
- But now we get a whole lot of other methods that are implemented in terms of these methods