Lecture 6: UML, Numbers, and Iterables

MPCS 51042-2 : Python Programming

Ron Rahaman

The University of Chicago, Dept of Computer Science

Nov 11, 2019

Table of Contents

Table of Contents

Classes and Instances

- A **class** defines a group of:
  - Attributes: Data
  - Methods: Functions

Attributes and Methods

- Discussed two types of attributes:
  - Instance attributes: Unique to each instance
  - Object attributes: Shared by all instances of a given class
- Discussed three kinds of methods:
  - Instance methods: Accessible via an instance. Can read/modify both instance and class attributes.
  - Class methods: Accessible via a class or instance. Can read/modify class attributes but not instance attributes.
  - Static methods: Accessible via a class or instance. Cannot read/write class or instance or class attributes.

Namespaces

- Assignment to a qualified names (`obj.X = 'foobar'`)
  - If the name `X` exists in the namespace of `obj`, then `obj.X` now refers to `'foobar'`
  - If not, creates the new name `obj.X` that refers to `'foobar'`
- Reference to a qualified name `obj.X`
  - Search for the name `X` in the following namespaces
    1. Instance
    2. Class
    3. All superclasses
  - If `X` is not found in any of those namespaces, raises a `NameError`
- **Qualified assignments and references never search surrounding scopes!**

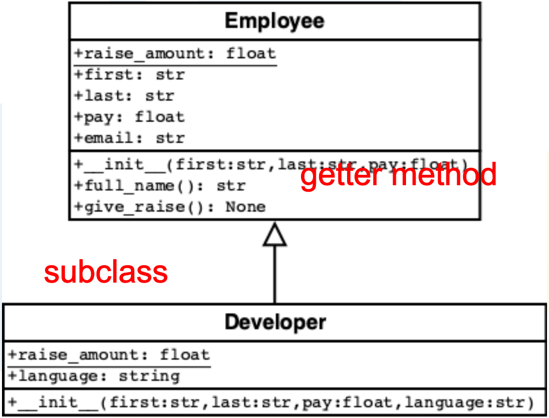
Table of Contents

Class Diagrams in UML

- Universal Modeling Language (UML) is used to represent many concepts in software engineering
- References:
  - Wikipedia: `https://en.wikipedia.org/wiki/Class_diagram`
  - IBM Developer: `https://developer.ibm.com/articles/the-class-diagram/`

Inheritance, Overriding, Extending: `Employee`

- Attributes
  - name : object_type
  - +/− for public/private
  - Thin line separates class (top) and instance (bottom) attributes
- Methods
  - name(arg_types) : return_type=default
- Inherit: listed in superclass only
- Override: listed in both superclass and subclass
- Extend: listed in subclass only

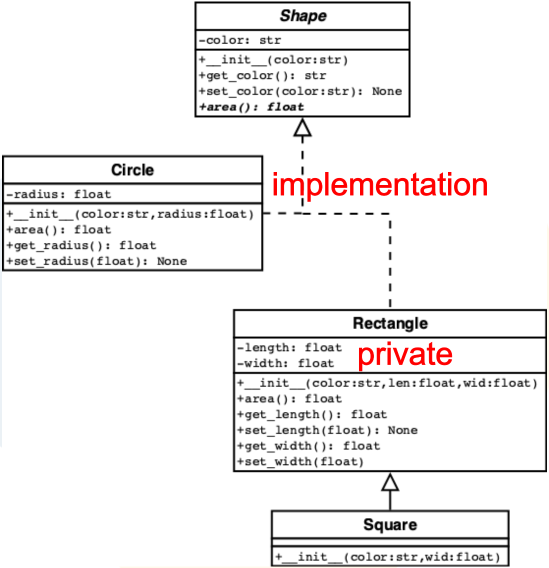


other method is inherited

constructer is overritten

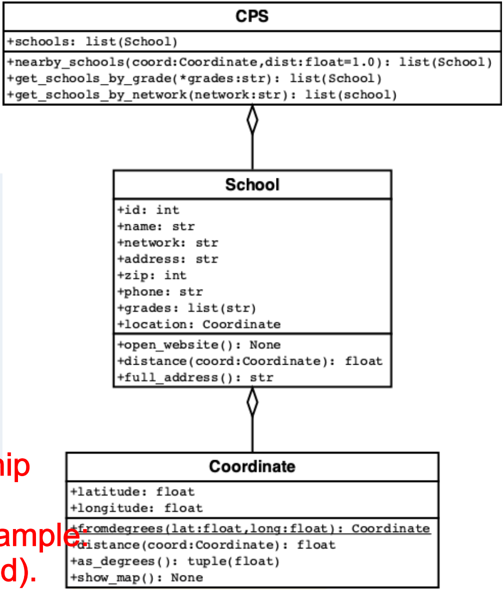
Implementation: The `Shape` Classes

- Implementation of an ABC is shown by a dashed line with an open arrowhead
- Notation varies for ABC name and abstract methods



Aggregation: The CPS Classes

- Aggregation means that one instance of a class contains one or more instances of the another
- It is shown by a solid line with an open diamond
- For example:
  - One School contains one Coordinate
  - One CPS contains a list of multiple Schools



Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still

Table of Contents

Version 1: Limited Functionality

```python
class Vector2D:
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)
```

String representation isn't user-friendly

```python
>>> a = Vector2D(10, 11)
>>> print(a)
<__main__.Vector2D object at 0x101ab2eb8>
```

Equality falls-back to identity (true if names refer to the same object)

```python
>>> b = Vector2D(10, 11)
>>> a == b
False
>>> c = a
>>> a == c
True
```

Version 2: Convenience Methods

```python
class Vector2D:
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def str(self):
        return "({}, {})".format(self.x, self.y)

    def eq(self, other):
        return self.x == other.x and self.y == other.y
```

Version 2: Convenience Methods

More convenient printing:

```
>>> a = Vector2D(10, 11)
>>> print(a.str())
(10.0, 11.0)
```

Equality works the way you'd expect:

```
>>> b = Vector2D(10, 11)
>>> a.eq(b)
True
```

Version 3: String Formattiong

```
class Vector2D:

    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __str__(self):    for print
        return "({}, {})".format(self.x, self.y)
                          for fallback to debug
    def __repr__(self):
        return "Vector2D{}".format(self)

    def eq(self, other):
        return self.x == other.x and self.y == other.y
```

Version 3: String Formatting

The `__str__` method is used by `print()` and `format()`. Expected to return a string. (`https://docs.python.org/3/reference/datamodel.html#object.__str__`)

```
>>> a = Vector2D(10, 11)
>>> L = ['foobar', a, -99]
>>> for x in L:
...     print(x)
foobar
(10.0, 11.0)
-99
>>> "I made a vector like: {}".format(a)
'I made a vector like: (10.0, 11.0)'
```

Version 3: String Formatting

The `__repr__` method is used for debugging and as a fallback for `__str__`. Should look like a valid constructor call (https://docs.python.org/3/reference/datamodel.html#object.__repr__)

```
>>> repr(a)
'Vector2D(10.0, 11.0)'
```

Table of Contents

Operator Overloading

- Operator overloading allows you to implement custom behavior for operators like $+$, $/$, $==$, etc.
- Expected to return **a new instance** of an object (not in-place)

c = [1]+[2] create a new instance,
even if it is mutable. List

Table of Contents

Unary Operators

You write...	Python executes ...
`-x`	`x.__neg__()`
`~x`	`x.__invert__()`
`abs(x)`	`x.__abs__()`

implement multiple methods within the same class that use the same name but a different set of parameters. That is called method overloading and represents a static form of polymorphism.

Version 4: Negative and Abs

```python
class Vector2D:
    def __abs__(self):
        return math.sqrt(self.x**2 + self.y**2)

    def __neg__(self):
        return Vector2D(-self.x, -self.y)
```

```python
>>> a = Vector2D(3, -4)
>>> -a
Vector2D(-3.0, 4.0)
>>> a
Vector2D(3.0, -4.0)

>>> abs(a)
5.0
>>> abs(-a)
5.0
```

Table of Contents

Infix Operators for Emulating Numeric Types

You write...	Python executes ...
`x + y`	`x.__add__(y)`
`x - y`	`x.__sub__(y)`
`x * y`	`x.__mul__(y)`
`x / y`	`x.__truediv__(y)`
`x // y`	`x.__floordiv__(y)`
`x ** y`	`x.__pow__(y)`

Reference: https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types

Bitwise/Logical Infix Operators

You write...	Python executes ...
`x & y`	`x.__and__(y)`
`x | y`	`x.__or__(y)`
`x ^ y`	`x.__xor__(y)`

Reference: `https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types`

Version 5: Vector and Scalar Addition

```python
class Vector2D:
    def __add__(self, other):
        if isinstance(other, Vector2D):
            return Vector2D(self.x + other.x, self.y + other.y)
        else:
            return Vector2D(self.x + other, self.y + other)
```

```python
>>> a = Vector2D(2, 5)
>>> b = Vector2D(1, -1)
>>> a + b
Vector2D(3.0, 4.0)
>>> abs(a + b)
5.0
>>> a + 10
Vector2D(12.0, 15.0)
```

Problems with Operand Order

```
>>> a = Vector2D(2, 5)
>>> b = 10
>>> a + b
Vector2D(12.0, 15.0)
>>> b + a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'Vector2D'
```

- `a.__add__(b)` is implemented for the types of a and b
- `b.__add__(a)` is not

solved by right-hand operators

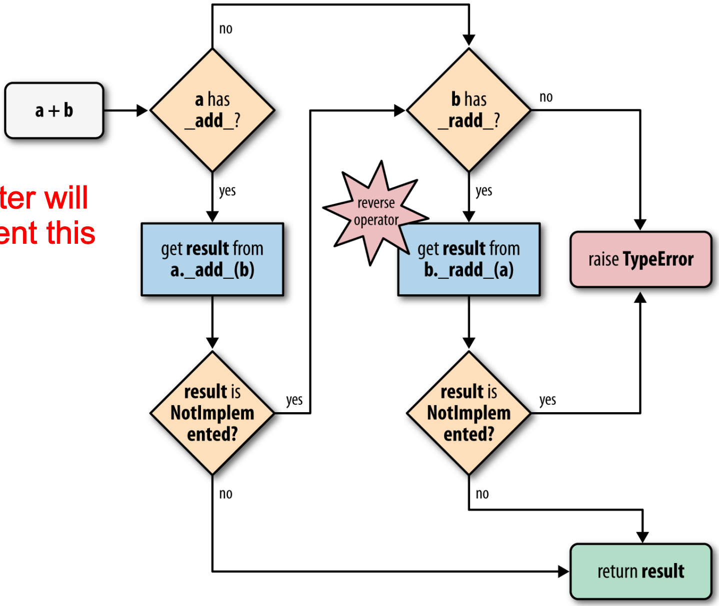
Reverse/reflected/right-hand Operators

You write...	Python executes ...
`x + y`	`y.__radd__(x)`
`x - y`	`y.__rsub__(x)`
`x * y`	`y.__rmul__(x)`
`x / y`	`y.__rtruediv__(x)`
`x // y`	`y.__rfloordiv__(x)`
`x ** y`	`y.__rpow__(x)`
`x & y`	`y.__rand__(x)`
`x | y`	`y.__ror__(x)`
`x ^ y`	`y.__rxor__(x)`

Reference: `https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types`

Dispatching Mechanism for Infix Operators

interpreter will
implement this



Version 6: Infix Operator Dispatching

a+b to b+a ?

```
from numbers import Real

class Vector2D:

    def __add__(self, other):
        if isinstance(other, Vector2D):
            return Vector2D(self.x + other.x, self.y + other.y)
        elif isinstance(other, Real):
            return Vector2D(self.x + other, self.y + other)
        else:
            return NotImplemented

    def __radd__(self, other):
        return self.__add__(other)
```

- Uses the `Real` ABC from `numbers`: `https://docs.python.org/3.7/library/numbers.html`
- Uses `isinstance()` instead of `type()`
- Uses `return NotImplemented` instead of `raise NotImplementedError`

Version 6: Infix Operator Dispatching

```
>>> Vector2D(2, 5) + Vector2D(1, -1)
Vector2D(3.0, 4.0)

>>> Vector2D(2, 5) + 1.5
Vector2D(3.5, 6.5)

>>> 1.5 + Vector2D(2, 5)
Vector2D(3.5, 6.5)

>>> Vector2D(2, 5) + '1.5'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Vector2D' and 'str'
```

Exercise: Infix Operators

Implement the subtraction and multiplication operators for vector.

- For a vector and a scalar:
  - Both subtraction and multiplication should apply the scalar to each element and return a new Vector (like in our addition operation)
- For a vector and vector:
  - Subtraction should subtract one vector from the other and return a new Vector (like addition)
  - Multiplication should perform a dot product and return a scalar

Table of Contents

Augmented Assignment Operators

You write...	Python executes ...
`x += y`	`x.__iadd__(y)`
`x -= y`	`x.__isub__(y)`
`x *= y`	`x.__imul__(y)`
`x /= y`	`x.__itruediv__(y)`
`x //= y`	`x.__ifloordiv__(y)`
`x **= y`	`x.__ipow__(y)`
`x &= y`	`x.__iand__(y)`
`x |= y`	`x.__ior__(y)`
`x ^= y`	`x.__ixor__(y)`

- Unlike other operators, should attempt in-place change
- If an augmented assignment operator is not implemented, the interpreter uses the infix operator (`x = x + y`, etc.)
- Reference: `https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types`

Table of Contents

Rich Comparison Operators

Operation	Forward method	Reverse method	Fallback
`x == y`	`x.__eq__(y)`	`y.__eq__(x)`	`return id(x) == id(b)`
`x != y`	`x.__ne__(y)`	`y.__ne__(x)`	`return not (a == b)`
`x > y`	`x.__gt__(y)`	`y.__lt__(x)`	`raise TypeError`
`x < y`	`x.__lt__(y)`	`y.__gt__(x)`	`raise TypeError`
`x >= y`	`x.__ge__(y)`	`y.__le__(x)`	`raise TypeError`
`x <= y`	`x.__le__(y)`	`y.__ge__(x)`	`raise TypeError`

Operator dispatching is the same as infix operators, except `==` and `!=` have a different fallback.

1. If the forward method returns `NotImplemented`, then the reverse method is called
2. Then if the reverse method returns `NotImplemented`, the fallback action if finally done.

Version 7: Equality

```python
class Vector2D:
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

```python
>>> a = Vector2D(2, 5)
>>> b = Vector2D(2, 5)
>>> a == b
True

>>> Vector2D(2, 5) == Vector2D(3,4)
False

>>> Vector2D(2, 5) != Vector2D(3,4)
True

>>> Vector2D(2, 5) + Vector2D(1, -1) == Vector2D(3, 4)
True
```

Exercise: Comparison Operators

- Implement the $<$, $>$, $\leq$, and $\geq$ operations for `Vector2D`
- To do this, compare the length of the vector using `abs()`

Table of Contents

Implementing a Numeric ABC?

- The numeric ABCs from the `numbers` module are fully specified in PEP 3141 (`https://www.python.org/dev/peps/pep-3141/`)
- Since the ABCs have many abstract methods, they are fully-implemented less often than the collection ABCs
- It is common to implement a subset of the numeric methods that make sense for your class (rather than fully implementing the numeric ABC)

Table of Contents

A Sequence of words in a sentence

```python
from collections.abc import Sequence
import reprlib
import re

class Sentence(Sequence):

    def __init__(self, text):
        self.text = text
        # self.words = text.split()
        self.words = re.findall(r'\w+', text)

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

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

    def __repr__(self):
        # return 'Sentence({})'.format(self.text)
        return 'Sentence({})'.format(reprlib.repr(self.text))
```

Using a Sentence as an iterable

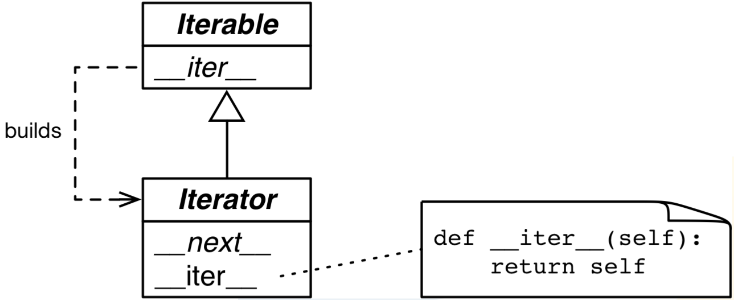
```
>>> s = Sentence('"The time has come", the Walrus said,')
>>> print(s)
Sentence('"The time ha... Walrus said,')
>>> for word in s:
...     print(word)
...
The
time
has
come
the
Walrus
said
>>> set([word.lower() for word in s])
{'said', 'time', 'has', 'the', 'walrus', 'come'}
```

Why can we use `Sentence` as an iterable?

When the interpreter iterates over an object `x`, it first calls `iter(x)`. A call to `iter` does the following:

1. If the object implements `__iter__`, then call it to obtain an iterator
2. Else if `__getitem__` is implemented, fetch items until `IndexError` is raised
3. Else raise `TypeError`

Protocols for `Iterable` and `Iterator`



- An `Iterable` ...
  - Implements `__iter__()`, which builds an `Iterator`
- An `Iterator` ...
  - Implements `__next__()`, which returns the next item or raises `StopIteration` when there are no more items
  - Implements `__item__()`, which returns itself

__iter__()

What happens in a `for` loop?

Consider `for i in obj`, where `obj` is iterable

1. Calls `obj.__iter__()` to obtain an iterator
2. Repeatedly calls the iterator's `__next__()` method and assigns the result to `i`
3. Break out of the loop when `StopIteration` is raised

Iterable is an object, which one can iterate over. It generates an Iterator when passed to iter() method.
Iterator is an object, which is used to iterate over an iterable object using __next__() method. Iterators have __next__() method, which returns the next item of the object.

Version 2: Using `Iterable` and `Iterator`

```
from collections.abc import Iterable

class Sentence(Iterable):
    def __init__(self, text):
        self.text = text
        self.words = re.findall(r'\w+', text)

    def __repr__(self):
        return 'Sentence({})'.format(reprlib.repr(self.text))

    def __iter__(self):
        return SentenceIterator(self.words) build an iterator
```

- `__iter__()` returns a new instance of an `Iterator`
- This supports multiple iterations on the same `Iterable`

Version 2: Using `Iterable` and `Iterator`

```
from collections.abc import Iterator

class SentenceIterator(Iterator):
    def __init__(self, words):
        self.words = words
        self.index = 0

    def __next__(self):
        if self.index < len(self.words):
            res = self.words[self.index]
            self.index += 1
            return res
        else:
            raise StopIteration

    def __iter__(self):
        return self
```

- This correctly implements an `Iterator`
- However, it is better practice to use a generator ...

Version 3: `__iter__()` builds a generator

```
from collections.abc import Iterable

class Sentence(Iterable):
    def __init__(self, text):
        self.text = text
        self.words = re.findall(r'\w+', text)

    def __repr__(self):
        return 'Sentence({})'.format(reprlib.repr(self.text))

    def __iter__(self):
        for w in self.words:
            yield w
        return
```

- `__iter__` is a generator function. When called, it builds an instance of a generator object.
- A generator object implements the iterator protocol
- No need for a separately-defined iterator class

Further (not required) Reading

An instance is an object in memory. Basically you create object and instantiate them when you are using them.

From Ramalho Ch 14:

- Version 4: A Lazy Iterable
- Version 5: Generator Expressions

From standard library:

- `itertools`: Efficient functions for working with (lazy) iterables (`https://docs.python.org/3.7/library/itertools.html`)