9.1. `numbers` — Numeric abstract base classes

Source code: Lib/numbers.py

The `numbers` module (**PEP 3141**) defines a hierarchy of numeric abstract base classes which progressively define more operations. None of the types defined in this module can be instantiated.

class `numbers.`**`Number`**

The root of the numeric hierarchy. If you just want to check if an argument *x* is a number, without caring what kind, use `isinstance(x, Number)`.

9.1.1. The numeric tower

class `numbers.`**`Complex`**

Subclasses of this type describe complex numbers and include the operations that work on the built-in `complex` type. These are: conversions to `complex` and `bool`, `real`, `imag`, `+`, `-`, `*`, `/`, `abs()`, `conjugate()`, `==`, and `!=`. All except `-` and `!=` are abstract.

`real`

Abstract. Retrieves the real component of this number.

`imag`

Abstract. Retrieves the imaginary component of this number.

abstractmethod **`conjugate`**()

Abstract. Returns the complex conjugate. For example, `(1+3j).conjugate() == (1-3j)`.

class `numbers.`**`Real`**

To `Complex`, `Real` adds the operations that work on real numbers.

In short, those are: a conversion to `float`, `math.trunc()`, `round()`, `math.floor()`, `math.ceil()`, `divmod()`, `//`, `%`, `<`, `<=`, `>`, and `>=`.

Real also provides defaults for `complex()`, `real`, `imag`, and `conjugate()`.

class `numbers.`**`Rational`**

Subtypes `Real` and adds `numerator` and `denominator` properties, which should be in lowest terms. With these, it provides a default for `float()`.

`numerator`

Abstract.

`denominator`

Abstract.

class `numbers.`**`Integral`**

Subtypes `Rational` and adds a conversion to `int`. Provides defaults for `float()`, `numerator`, and `denominator`. Adds abstract methods for `**` and bit-string operations: `<<`, `>>`, `&`, `^`, `|`, `~`.

9.1.2. Notes for type implementors

Implementors should be careful to make equal numbers equal and hash them to the same values. This may be subtle if there are two different extensions of the real numbers. For example, `fractions.Fraction` implements `hash()` as follows:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

9.1.2.1. Adding More Numeric ABCs

There are, of course, more possible ABCs for numbers, and this would be a poor hierarchy if it precluded the possibility of adding those. You can add `MyFoo` between `Complex` and `Real` with:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

9.1.2.2. Implementing the arithmetic operations

We want to implement the arithmetic operations so that mixed-mode operations either call an implementation whose author knew about the types of both arguments, or convert both to the nearest built in type and do the operation there. For subtypes of `Integral`, this means that `__add__()` and `__radd__()` should be defined as:

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

There are 5 different cases for a mixed-type operation on subclasses of `Complex`. I'll refer to all of the above code that doesn't refer to `MyIntegral` and `OtherTypeIKnowAbout` as "boilerplate". `a` will be an instance of `A`, which is a subtype of `Complex` (`a : A <: Complex`), and `b : B <: Complex`. I'll consider `a + b`:

1. If `A` defines an `__add__()` which accepts `b`, all is well.
2. If `A` falls back to the boilerplate code, and it were to return a value from `__add__()`, we'd miss the possibility that `B` defines a more intelligent `__radd__()`, so the boilerplate should return `NotImplemented` from `__add__()`. (Or `A` may not implement `__add__()` at all.)
3. Then `B`'s `__radd__()` gets a chance. If it accepts `a`, all is well.
4. If it falls back to the boilerplate, there are no more possible methods to try, so this is where the default implementation should live.

5. If `B <: A`, Python tries `B.__radd__` before `A.__add__`. This is ok, because it was implemented with knowledge of `A`, so it can handle those instances before delegating to `Complex`.

If `A <: Complex` and `B <: Real` without sharing any other knowledge, then the appropriate shared operation is the one involving the built in `complex`, and both `__radd__()` s land there, so `a+b == b+a`.

Because most of the operations on any given type will be very similar, it can be useful to define a helper function which generates the forward and reverse instances of any given operator. For example, `fractions.Fraction` uses:

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```