Data-directed programming

- A fully generic approach to implementing of generic arithmetic
  - apply ***arbitrary operators*** to ***arbitrary types*** by using a *dictionary* to store implementations of *various combinations*

Our example: apply

- We can implement both generic *addition* and *multiplication* without redundant logic:

```
>>> def apply(operator_name, x, y):
        tags = (type_tag(x), type_tag(y))
        key = (operator_name, tags)
        return apply.implementations[key](x, y)
```

KEY is constructed from the operator name (e.g., 'add') and a tuple of type tags for the arguments

Our example: population

Support for *multiplication* on complex and rational numbers:

```
>>> def mul_complex_and_rational(z, r):
        return ComplexMA(z.magnitude * r.numer / r.denom, z.angle)
>>> mul_rational_and_complex = lambda r, z:
        mul_complex_and_rational(z, r)
>>> apply.implementations = {('mul', ('com', 'com')): mul_complex,
                             ('mul', ('com', 'rat')): mul_complex_and_rational,
                             ('mul', ('rat', 'com')): mul_rational_and_complex,
                             ('mul', ('rat', 'rat')): mul_rational}
```

סדר הפוך

Our example: population

- Add the addition implementations from *add* to *apply*:

```
>>> adders = add.implementations.items()
>>> apply.implementations.update(
                {('add', tags):fn for (tags, fn) in adders})
```

Final result

apply supports 8 different implementations in a single table, we can use it to manipulate rational and complex numbers quite generically:

```
>>> apply('add', ComplexRI(1.5, 0), Rational(3, 2))
ComplexRI(3.0, 0)

>>> apply('mul', Rational(1, 2), ComplexMA(10, 1))
ComplexMA(5.0, 1)
```