Designing Types

- **Type**: set of values and the operations on them
  - `int`: set of integers; `ops`: `+`, `*`, `/`, `…`
  - `Time`: set of times of day; `ops`: time span, before/after, `…`
  - `Worker`: set of all possible workers; `ops`: hire, pay, promote, `…`
  - `Rectangle`: set of all axis-aligned rectangles in 2D; `ops`: contains, intersect, `…`

- To define a class, think of a *real type* you want to make
  - Python gives you the tools, but does not do it for you
  - Physically, any object can take on any value
  - Discipline is required to get what you want

Making a Class into a Type

1. Think about what values you want in the set
   - What are the attributes? What values can they have?
2. Think about what operations you want
   - This often influences the previous question
3. To make (1) precise: write a **class invariant**
   - Statement we promise to keep true after every method call
4. To make (2) precise: write **method specifications**
   - Statement of what method does/what it expects (preconditions)
   - Write your code to make these statements true!

Planning out a Class

```python
class Time(object):
    """Instances represent times of day."
    Instance Attributes:
    hour: hour of day [int in 0..23]
    min: minute of hour [int in 0..59]"

    def __init__(self, hour, min):
        """The time is Noon or later."""
        Pre: hour in 0..23; min in 0..59"

    def increment(self, hours, mins):
        """Move this time <hours> hours and <mins> minutes into the future."
        Pre: hours >= 0; mins in 0..59"

    def isPM(self):
        """Returns: this time is noon or later."""
```

**Class Invariant**

States what attributes are present and what values they can have.
A statement that will always be true of any Time instance.

**Method Specification**

States what the method does.
Gives preconditions stating what is assumed true of the arguments.

Implementing an Initializer

```python
def __init__(self, hour, min):
    self.hour = hour
    self.min = min
```

This is to start

You put code here

This should be true at the end

Implementing a Method

```python
def increment(self, hours, mins):
    self.hour += hours
    self.min += mins
```

This is also to start

You put code here

This should be true at the end
Enforce Method Preconditions with `assert`

```python
class Time(object):
    """Instance attributes:
    hour: hour of day [int in 0..23]
    minute: minute of hour [int in 0..59]"

    def __init__(self, hour, minute):
        """The time hour:minute.
        Pre: hour in 0..23; minute in 0..59"
        self.hour = hour
        self.minute = minute

    def increment(self, hours, minutes):
        """Move this time <hours> hours
        and <minutes> minutes into the future.
        Pre: hours is < 24; minutes is < 60"
        self.hours += hours
        self.minutes += minutes

        if self.minutes >= 60:
            self.minutes -= 60
            self.hours += 1

        if self.hours >= 24:
            self.hours -= 24

        assert 0 <= self.hours and self.hours < 24
        assert 0 <= self.minutes and self.minutes < 60
```

**Enforcing Invariants**

```python
class Fraction(object):
    """Instance attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"

    def __init__(self, n, d):
        """Alters denominator to be d
        Pre: d is an int > 0"
        self.numerator = n
        self.denominator = d

    def getDenominator(self):
        """Returns: denominator"
        return self.denominator

    def setDenominator(self, d):
        """Sets denominator to value
        Pre: d is an int >= 0"

        if d > 0:
            self.denominator = d
            self.numerator = n
```

**Data Encapsulation**

```python
class Fraction(object):
    """Instance attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"

    def __init__(self, n, d):
        """Alters denominator to be d
        Pre: d is an int > 0"
        self.numerator = n
        self.denominator = d
```

**Mutable vs. Immutable Attributes**

<table>
<thead>
<tr>
<th>Mutable</th>
<th>Immutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can change value directly</td>
<td>Can’t change value directly</td>
</tr>
<tr>
<td>If class invariant met</td>
<td>May change “behind scenes”</td>
</tr>
<tr>
<td>Example: tcolor</td>
<td>Example: t.x</td>
</tr>
<tr>
<td>Has both getters and setters</td>
<td>Has only a getter</td>
</tr>
<tr>
<td>Setters allow you to change</td>
<td>No setter means no change</td>
</tr>
<tr>
<td>Enforce invariants w/ asserts</td>
<td>Getter allows limited access</td>
</tr>
</tbody>
</table>

**Hiding Methods From Access**

```python
class Fraction(object):
    """Instance attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"

    def _is_denominator(self, d):
        """Return: True if d valid denom
        inside of the same class"
        return type(d) == int and d > 0

    def _inh__(self, p=0, d=1):
        assert self._is_denominator(d)
        self.numerator = n
        self.denominator = d
```

**Data Encapsulation**

```python
class Fraction(object):
    """Instance attributes:
    numerator: top [int]
    denominator: bottom [int > 0]"
```

**Mutable vs. Immutable Attributes**

<table>
<thead>
<tr>
<th>Mutable</th>
<th>Immutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can change value directly</td>
<td>Can’t change value directly</td>
</tr>
<tr>
<td>If class invariant met</td>
<td>May change “behind scenes”</td>
</tr>
<tr>
<td>Example: tcolor</td>
<td>Example: t.x</td>
</tr>
<tr>
<td>Has both getters and setters</td>
<td>Has only a getter</td>
</tr>
<tr>
<td>Setters allow you to change</td>
<td>No setter means no change</td>
</tr>
<tr>
<td>Enforce invariants w/ asserts</td>
<td>Getter allows limited access</td>
</tr>
</tbody>
</table>

**Mutator Method**

```python
>>> f numerator = 5
Good:
>>> f.numerator = 5
```

**Getter Method**

```python
>>> x = 5 * f.numerator

Good:
>>> x = 5 * f.getNumerator()
```