Designing Types

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

### Planning out a Class

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

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

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

```
def increment(self, hours, mins):
    hours = hours % 24
    mins = mins % 60
    self.hour = (self.hour + hours) % 24
    self.min = (self.min + mins) % 60
```

### Planning out a Class

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

```
def __init__(self, t, l, b, r):
    self.t = t
    self.l = l
    self.b = b
    self.r = r
```

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

```
def area(self):
    """Return: area of the rectangle."
    return (self.r - self.l) * (self.t - self.b)
```

### Implementing a Class

- All that remains is to fill in the methods. (All?!)  
- When implementing methods:  
  1. Assume preconditions are true  
  2. Assume class invariant is true to start  
  3. Ensure method specification is fulfilled  
  4. Ensure class invariant is true when done  
- Later, when using the class:  
  1. When calling methods, ensure preconditions are true  
  2. If attributes are altered, ensure class invariant is true

### Implementing an Initializer

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

This is true to start
You put code here
This should be true at the end
**Implementing a Method**

- Instance variables:
  - hour: hour of day [int in 0..23]
  - min: minute of hour [int in 0..59]

  **This is true to start**
  - What we are supposed to accomplish

  **This is also true to start**
  - You put code here

- Instance variables:
  - hour: hour of day [int in 0..23]
  - min: minute of hour [int in 0..59]

  **This should be true at the end**

**Enforce Method Preconditions with assert**

- Class `Time(object):`
  - """Instances represent times of day."""

  **def __init__(self, hour, min):**
  - """The time hour:min.
  - Pre: hour in 0..23; min in 0..59"""
  - assert type(hour) == int
  - assert type(min) == int
  - assert 0 <= hour and hour < 24
  - assert 0 <= min and min < 60

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

- Example:
  - >>> h = 60*t = Time(2,45)
  - >>> t.min = 70

  **Invariant violation!**

- Nothing we can do
  - User can access instance attributes via assignment
    - Wrote methods assuming invariant always true
    - Our enforcement code is all in method definitions

**Data Encapsulation**

- **Idea:** Force the user to only use methods
- **Do not allow direct access of attributes**

  - **Setter Method**
    - Used to change an attribute
    - Replaces all assignment statements to the attribute
    - **Bad:**
      - >>> t.min = 55
    - **Good:**
      - >>> t.setMin(55)
  
  - **Getter Method**
    - Used to access an attribute
    - Replaces all usage of attribute in an expression
    - **Bad:**
      - >>> h = 60*t.min
    - **Good:**
      - >>> h = 60*t.getMin()