Designing types

- One definition of a *type*: **a set of objects with the operations on those objects**.
  - int—set: integers; ops: +, –, *, /, …
  - Time—set: times of day; ops: time span, before/after, …
  - Worker—set: all possible workers; ops: hire, pay, promote, …
  - Rectangle—set: all axis-aligned rectangles in 2D; ops: contains, intersect, …
- When you define a class, you are (should be) thinking of a "real type" you want to create
  - Python gives you the tools to do this, but doesn't do it for you
  - Physically, any object can take on any value
  - Discipline is required to get what you want

Making a class a "real type"

1. Think about what values you want in the set
   - What attributes? What values can they have?
2. Think about what operations you want
   - Often influences the previous question

- To make (1) precise: write a *class invariant*
  - A statement we promise ourselves to keep true after every method call
- To make (2) precise: write specifications of methods
  - A statement of what the method does and what it expects (preconditions)
- Write your code to make these statements true!

Planning out a class

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

    def __init__(self, hour, min):
        """The time hour:min.
        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 [int] >= 0; mins in 0..59"""

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

integer ranges imply hour, min are ints

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

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

Planning out a class

```
class Rectangle(object):
    """Instances represent rectangular
    regions of the plane.
    Instance variables:
        t [float]: y coordinate of top edge
        l [float]: x coordinate of left edge
        b [float]: y coordinate of bottom edge
        r [float]: x coordinate of right edge
    For all Rectangles, l <= r and b <= t.
    """

    def __init__(self, t, l, b, r):
        """The rectangle [l, r] x [t, b]
        Pre: args are floats; l <= r; b <= t"""

    def area(self):
        """Return: area of the rectangle."""

    def intersection(self, other):
        """Return: new Rectangle describing
        intersection of self with other."""
```

class invariant
States what attributes are present and what values they can have.
A statement that will always be true of Rectangle instances.

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

Planning out a class

```
class Hand(object):
    """Instances represent a hand in cards.
    Instance variables:
        cards [list of Card]: cards in the hand
    This list is sorted according to the
    ordering defined by the Card class.
    """

    def __init__(self, deck, n):
        """Draw a hand of n cards.
        Pre: deck is a list of >= n cards"""

    def is_full_house(self):
        """Return: True if this hand is a full
        house."""

    def discard(self, k):
        """Discard the k-th card."""
```

class invariant
States what attributes are present and what values they can have.
A statement that will always be true of Hand instances.

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

Implementing a class

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

Implementing the Time initializer

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

This is true to start

```
self.hour = hour
self.min = min
```

You put code here

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

This should be true at the end

Implementing the Rectangle initializer

```
def __init__(self, l, r, b, t):
    """The rectangle [l, r] x [b, t]
    Pre: args are floats; l <= r; b <= t"""
```

This is true to start

```
self.l = l
self.r = r
self.b = b
self.t = t
```

You put code here

```
Instance variables:
    l [float]: x coord. of left edge
    r [float]: x coord. of right edge
    b [float]: y coord. of bottom edge
    t [float]: y coord. of top edge
For all Rectangles, l <= r and b <= t.
```

This should be true at the end

Implementing a Card initializer

```
def __init__(self, code):
    """The card specified by a two-character
    string; the first character gives the rank, and
    the second, the suit. "3H" is the three of
    Hearts; "QS" is the Queen of Spades.
    Pre: code is a two-character string; code[0] is
    in 'A23456789TJQK'; code[1] is in 'CDHS'."""
```

This is true to start

```
suit = 'CDHS'.index(code[1])
rank = ' A23456789TJQK'.index(code[0])
```

You put code here

```
Instance variables:
    suit [int in 0..NUM_SUITS-1]:
        the index of this card's suit
    rank [int in 1..NUM_RANKS]:
        the index of this card's rank
```

This should be true at the end

Implementing a Time method

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

This is true to start

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

What we are supposed to accomplish

This is also true to start

```
self.min = self.min + min
self.hour = (self.hour + hour +
                 self.min / 60)
self.min = self.min % 60
self.hour = self.hour % 24
```

You put code here

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

This should be true at the end

The view from outside

- Invariants and preconditions serve two purposes
- They are tools for you, as the author, to think through your plans in a disciplined way
- They communicate to the user* of the class how they are allowed to use it
- Together they are the *interface* of the class
  - interface between two programmers
  - interface between two parts of the program

* …who might well be you!

in•ter•face |ˈintərˌfās| noun

1. a point where two systems, subjects, organizations, etc., meet and interact : the interface between accountancy and the law.
   - *chiefly Physics* a surface forming a common boundary between two portions of matter or space, e.g., between two immiscible liquids : the surface tension of a liquid at its air/liquid interface.
2. *Computing* a device or program enabling a user to communicate with a computer.
   - a device or program for connecting two items of hardware or software so that they can be operated jointly or communicate with each other.

—The Oxford American Dictionary

Acquiring more "real type" behavior

- Want two different objects to be ==?
  - define `__eq__(self, other)`
    true if and only if self == other
  - define `__ne__(self, other)`
    true if and only if self != other
- Want your type to be ordered?
  - define `__cmp__(self, other)`
    negative if self < other;
    zero if self == other;
    positive if self > other
  - or define `__eq__`, `__ne__`, `__lt__`, `__gt__`, `__le__`, `__ge__`

is vs. ==

"a is b" means that a and b *are the same object* (same ID).
Cannot be customized.
Opposite: "a is not b"
Use for "is None", "is not None"

"a == b" means that a and b *have the same value*.
Can be customized by defining special methods.
Opposite: "a != b"