Example: Converting Values to Strings

str() Function

- **Usage**: str(<expression>)
  - Evaluates the expression
  - Converts it into a string
- How does it convert?
  - str(2) → '2'
  - str(True) → 'True'
  - str('True') → 'True'
  - str(Point3()) → '(0.0,0.0,0.0)'

repr() Function

- **Usage**: repr(<expression>)
  - Evaluates the expression
  - Converts it into a string
- How does it convert?
  - repr(2) → '2'
  - repr(True) → 'True'
  - repr('True') → "'True'"
  - repr(Point3()) → "<class 'Point3'> (0.0,0.0,0.0)"

What Does str() Do On Objects?

- Does **NOT** display contents

```
>>> p = Point3(1,2,3)
>>> str(p)
'<Point3 object at 0x1007a90>'
```

- Must add a special method
  - __str__ for str()
  - __repr__ for repr()
- Could get away with just one
  - repr() requires __repr__
  - str() can use __repr__ (if __str__ is not there)

```
class Point3(object):
    """Class for points in 3d space"""
    …
    def __str__(self):
        """Returns: string with contents"""
        return '('+self.x + ',' +
                    self.y + ',' +
                    self.z + ')'

    def __repr__(self):
        """Returns: unambiguous string"""
        return str(self.__class__)+
               str(self)
```

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

- To make (1) precise: write a *class invariant*
  - Statement we promise to keep true **after every method call**
- 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

```
class Time(object):
    """Class to 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 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 is int >= 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.

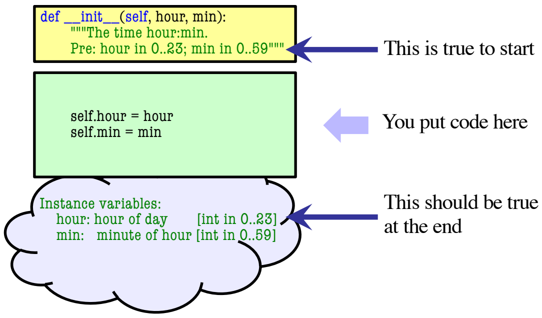
A statement that will always be true of any Time instance.

Method Specification

States what the method does.

Gives preconditions stating what is assumed true of the arguments.

Implementing an Initializer



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

This is true to start

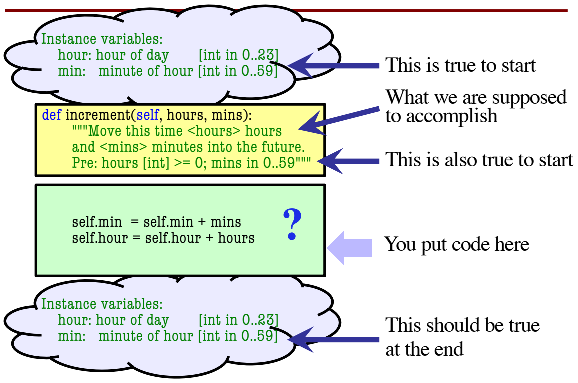
```
    self.hour = hour
    self.min = min
```

You put code here

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

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

```
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 + mins
    self.hour = self.hour + hours
```

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 0 <= hour and hour < 24
        assert type(min) == int
        assert 0 <= min and min < 60

    def increment(self, hours, mins):
        """Move this time <hours> hours
        and <mins> minutes into the future.
        Pre: hours is int >= 0; mins in 0..59"""
        assert type(hour) == int
        assert type (min) == int
        assert hour >= 0 and
        assert 0 <= min and min < 60
```

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

Initializer creates/initializes all of the instance attributes.

Asserts in initializer guarantee the initial values satisfy the invariant.

Asserts in other methods enforce the method preconditions.

Hiding Methods From Access

- Put underscore in front of a method will make it **hidden**
  - Will not show up in help()
  - But it is still there…
- Hidden methods
  - Can be used as **helpers** inside of the same class
  - But it is bad style to use them outside of this class
- Can do same for attributes
  - Underscore makes it hidden
  - Do not use outside of class

```
class Time(object):
    """INSTANCE ATTRIBUTES:
        hour: the hour   [int in 0..23]
        min: the minute [int in 0..59]"""

    def _is_minute(self,m):
        """Return: True if m valid minute"""
        return (type(m) == int and
                m >= 0 and m < 60)

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

HIDDEN

Helper method

Enforcing Invariants

```
class Time(object):
    """INSTANCE ATTRIBUTES:
      hour: the hour   [int in 0..23]
      min: the minute [int in 0..59]
    """
```

Invariants: Properties that are always true.

- These are just comments!

  `>>> t = Time(2,30)`

  `>>> t.hour = 'Hello'`
- How do we prevent this?

- **Idea**: Restrict direct access
  - Only access via methods
  - Use asserts to enforce them
- **Example**:

```
def getHour(self):
    """Returns: the hour"""
    return self.hour

def setHour (self,value):
    """Sets hour to value"""
    assert type(value) == int
    assert value >= 0 and value < 24
    self.numerator = value
```

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.hour = 5`
- **Good**:

  `>>> f.setHour(5)`

Getter Method

- Used to access an attribute
- Replaces all usage of attribute in an expression
- **Bad**:

  `>>> x = 3*t.hour`
- **Good**:

  `>>> x = 3*t.getHour()`

Data Encapsulation

```
class Time(object):
    """INSTANCE ATTRIBUTES:
        _hour: the hour   [int in 0..23]
        _min: the minute [int in 0..59]"""

    def getHour (self):
        """Returns: hour attribute"""
        return self._hour

    def setHour(self, h):
        """ Sets hour to h
        Pre: h is an int in 0..23"""
        assert type(h) == int
        assert 0 <= h and h < 24
        self._hour = d
```

Do this for all of your attributes

Getter

Setter

Naming Convention

The underscore means "should not access the attribute directly."

Precondition is same as attribute invariant.

Mutable vs. Immutable Attributes

Mutable

- Can change value directly
  - If class invariant met
  - **Example**: turtle.color
- Has both getters and setters
  - Setters allow you to change
  - Enforce invariants w/ asserts

Immutable

- Can't change value directly
  - May change "behind scenes"
  - **Example**: turtle.x
- Has only a getter
  - No setter means no change
  - Getter allows limited access

May ask you to differetiate on the exam