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 '('+str(self.x) + ',' +
                   str(self.y) + ',' +
                   str(self.z) + ')'

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

1

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!

2

Planning out a Class

```
class Time(object):
    """Class to represent times of day.

    Inv: hour is an int in 0..23
    Inv: min is an 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 time hours and mins
        into the future.
        Pre: hours int >= 0; mins in 0..59"""

    def isPM(self):
        """Returns: True if 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.

3

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

```
Inv: hour is an int in 0..23
Inv: min is an int in 0..59
```
This should be true at the end

4

Implementing a Method

```
Inv: hour is an int in 0..23
Inv: min is an 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

```
Inv: hour is an int in 0..23
Inv: min is an int in 0..59
```
This should be true at the end

5

Enforce Method Preconditions with assert

```
class Time(object):
    """Class to 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
        assert 0 <= min and min < 60
```

```
Inv: hour is an int in 0..23
Inv: min is an 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.

6

Hiding Methods From Access

- Hidden methods
  - start with an **underscore**
  - do not show up in help()
  - are meant to be **internal** (e.g. helper methods)
- But they are **not restricted**
  - You can still access them
  - But this is bad practice!
  - Like a precond violation
- Can do same for attributes
  - Underscore makes it hidden
  - Only used inside of methods

```
class Time(object):
    """Class to represent times of day.

    Inv: hour is an int in 0..23
    Inv: min is an 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)
        ...
```

Helper

7

Enforcing Invariants

```
class Time(object):
    """Class to repr times of day.

    Inv: hour is an int in 0..23
    Inv: min is an 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
```

8

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()
```

9

Data Encapsulation

```
class Time(object):
    """Class to repr times of day. """

    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
```

NO ATTRIBUTES in class specification

Getter

Method specifications describe the attributes

Setter

Setter precondition is same as the **invariant**

10

Encapsulation and Specifications

```
class Time(object):
    """Class to represent times of day. """

    ### Hidden attributes
    # Att _hour: hour of the day
    # Inv: _hour is an int in 0..23
    # Att _min: minute of the hour
    # Inv: _min is an int in 0..59
```

No attributes in class spec

These comments make it part of the **class invariant** but not part of the (public) **interface**

These comments do not go in help()

11

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 differentiate on the exam

12