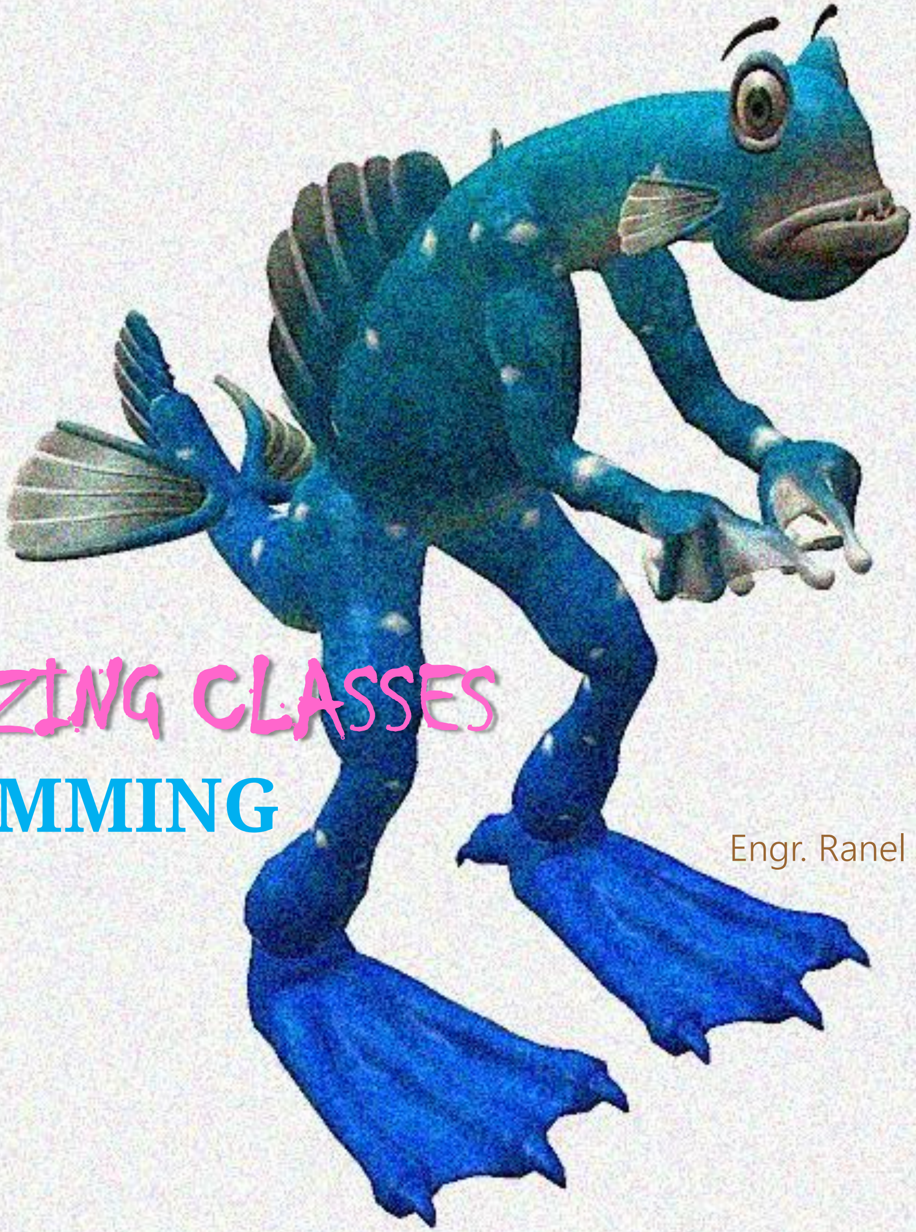
VII. CUSTOMIZING CLASSES

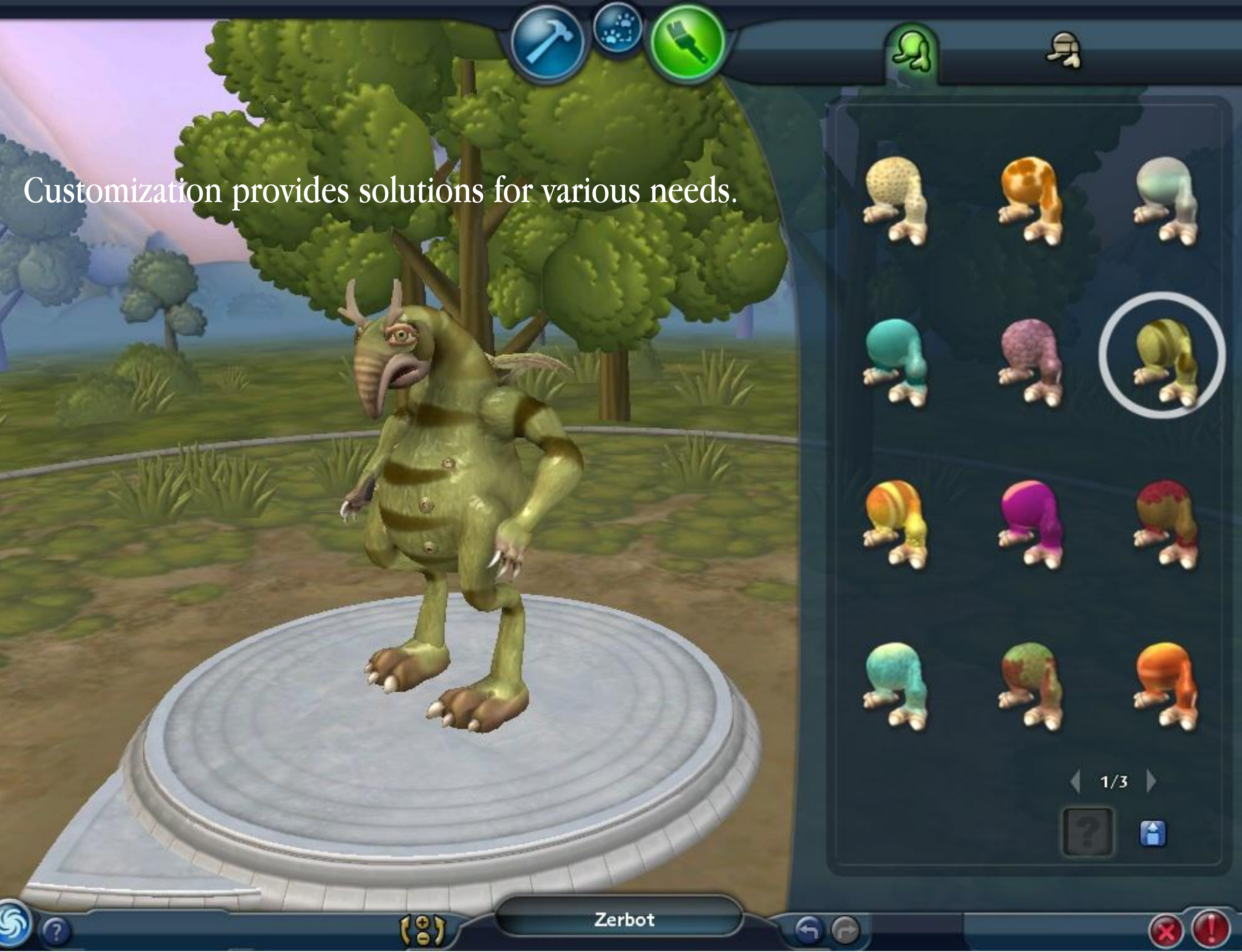
PYTHON PROGRAMMING

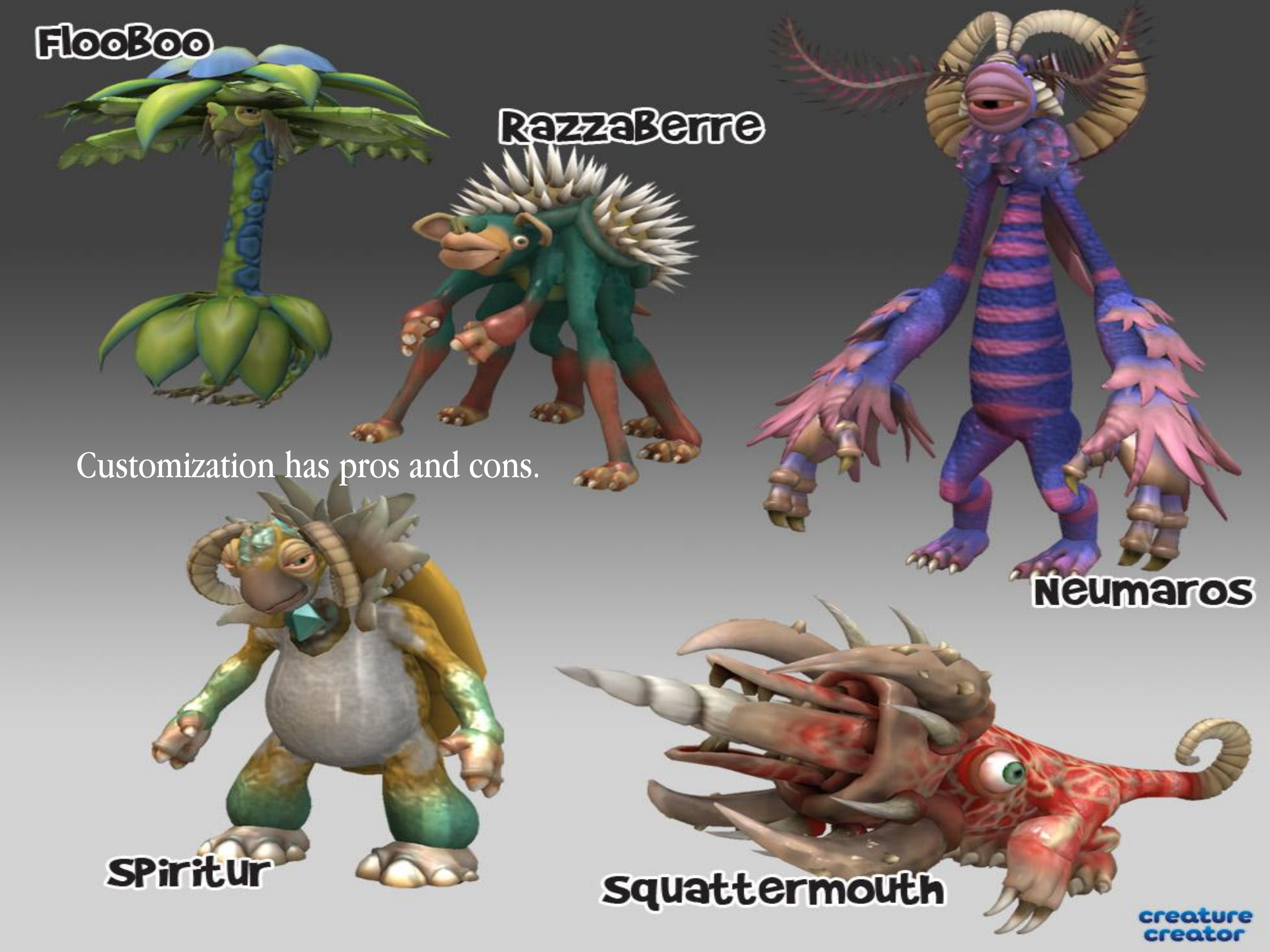


Engr. Ranel O. Padon

PYTHON PROGRAMMING TOPICS

Customization provides solutions for various needs.





Customization has pros and cons.

INTRODUCTION

Special Methods: __init__ and __del__

__init__ creates objects

__del__ destroys objects

INTRODUCTION

The typical method-call notation is cumbersome for mathematical classes.

>> polynomial1.add(polynomial2)

better or more natural way:

>> polynomial1 + polynomial2

This is called Operator Overloading.

INTRODUCTION

For faster development, reuse, modify, or extend
the built-in attributes, methods & operators of a Class.

INTRODUCTION

Python enables the programmer to overload most operators to be sensitive to the context in which they are used.

```
>> print 2 + 3
>> print "Mang" + "Jose"

>> print 2 * 3
>> print "Mang" * 2        #prints "MangMang"
```

INTRODUCTION

The interpreter takes the appropriate action based on the manner in which the operator is used.

INTRODUCTION

Some operators are overloaded frequently, especially operators like + and –.

The job performed by overloaded operators also can be performed by explicit method calls, but operator notation is often clearer.

STRING REPRESENTATION

A Python class can define special method __str__, to provide an informal (i.e., human-readable) string representation of an object of the class.

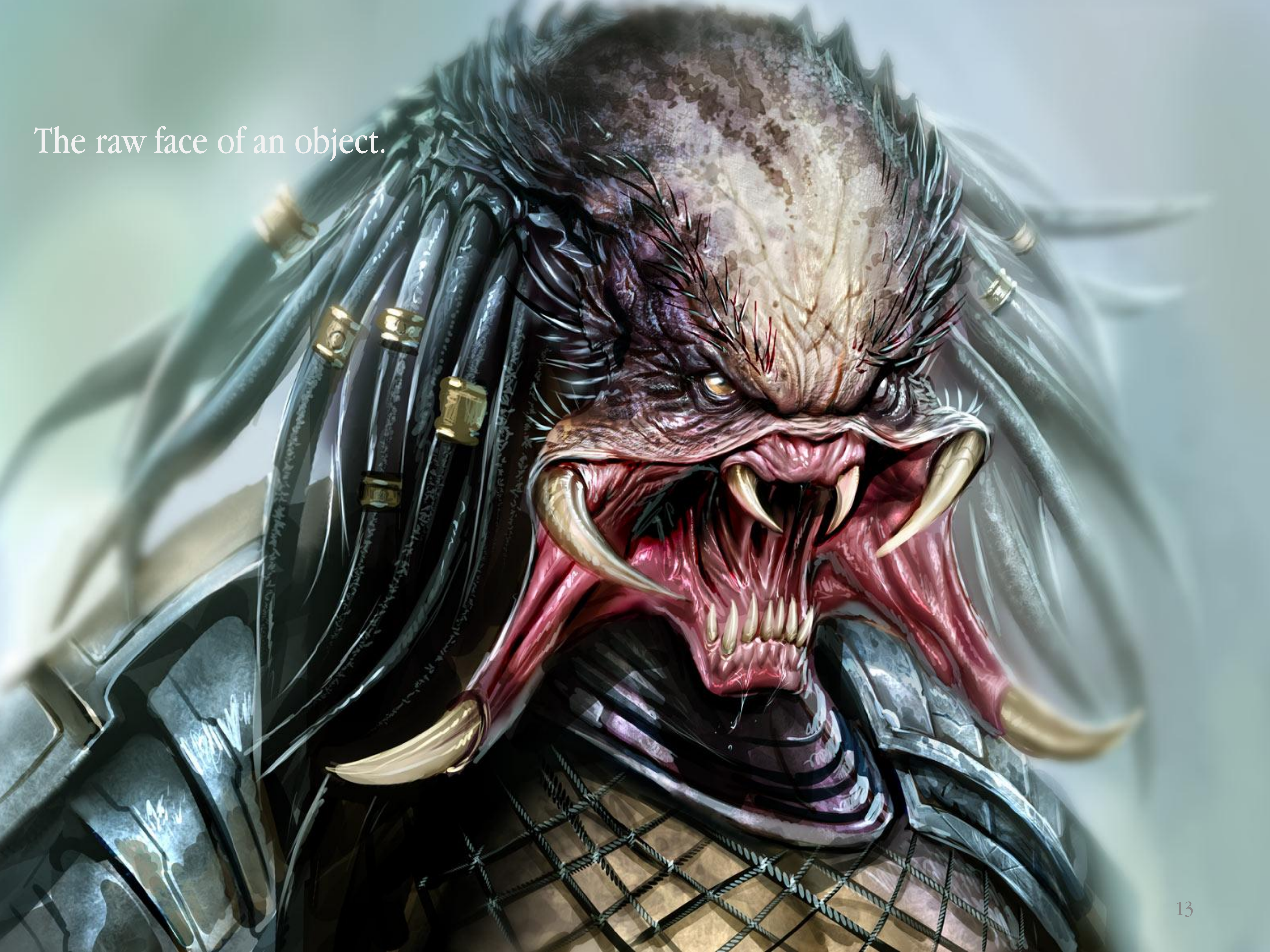
__str__ is analogous to Java's **toString()** method.

STRING REPRESENTATION

If a client program of the class contains the statement

print objectOfClass

Python calls the object's __str__ method and outputs the string returned by that method.


The raw face of an object.

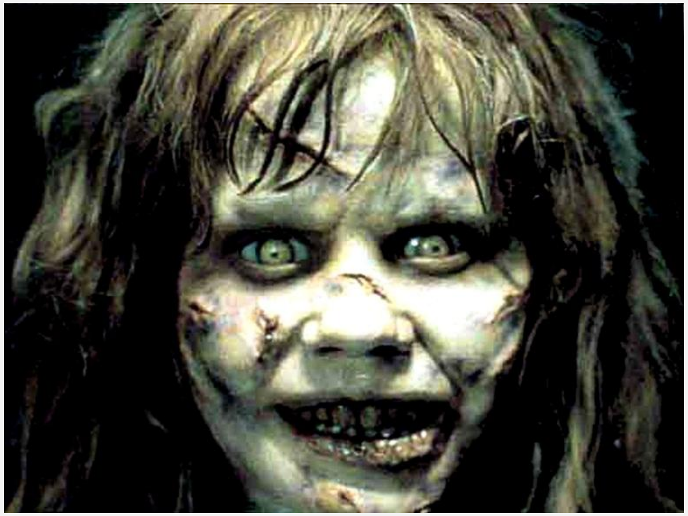
STRING REPRESENTATION

default invocation

```
>> print halimawSaBanga
    __main__.HalimawSaBanga instance at 0x0204E120
```

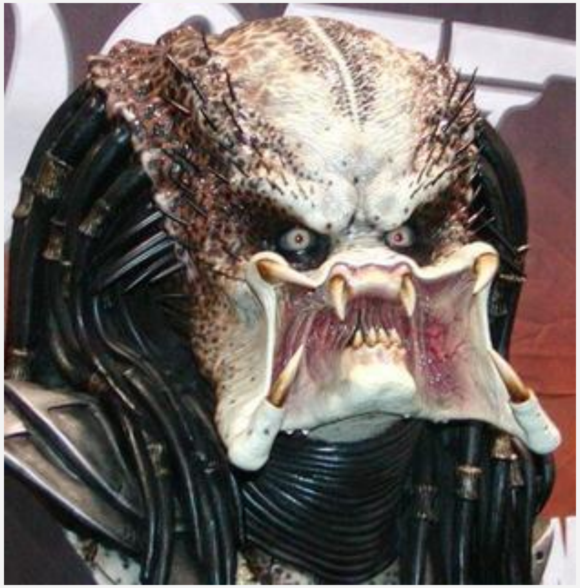
if __str__ is overriden

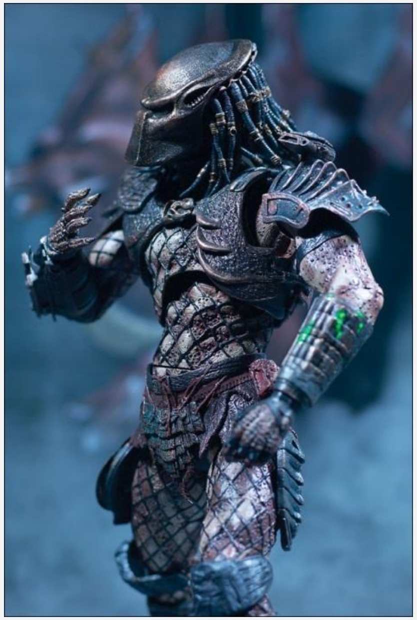
```
>> print halimawSaBanga
    500-year-old na Halimaw natagpuan sa banga!
```



The Raw Faces of an Object

The __str__ Faces of an Object





STRING REPRESENTATION

```
# Representation of phone number in USA format: (xxx) xxx-xxxx.

class PhoneNumber:
   """Simple class to represent phone number in USA format"""

   def __init__( self, number ):
      """Accepts string in form (xxx) xxx-xxxx"""

      self.areaCode = number[ 1:4 ]  # 3-digit area code
      self.exchange = number[ 6:9 ]  # 3-digit exchange
      self.line = number[ 10:14 ]  # 4-digit line

   def __str__( self ):
      """Informal string representation"""

      return "(%s) %s-%s" % \
         ( self.areaCode, self.exchange, self.line )
```

STRING REPRESENTATION

```
def test():

    # obtain phone number from user
    newNumber = raw_input(
        "Enter phone number in the form (123) 456-7890:\n" )

    phone = PhoneNumber( newNumber )  # create PhoneNumber object
    print "The phone number is:",
    print phone  # invokes phone.__str__()

if __name__ == "__main__":
    test()
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1234
The phone number is: (800) 555-1234
```

`print phone` = `print phone.__str__()`

ATTRIBUTE ACCESS

Method	Description
`__delattr__`	Executes when a client deletes an attribute (e.g., `del anObject.attribute`)
`__getattr__`	Executes when a client accesses an attribute name that cannot be located in the object's `__dict__` attribute (e.g., `anObject.unfoundName`)
`__setattr__`	Executes when a client assigns a value to an object's attribute (e.g., `anObject.attribute = value`)

ATTRIBUTE ACCESS

```
class Time:
   """Class Time with customized attribute access"""

   def __init__( self, hour = 0, minute = 0, second = 0 ):
      """Time constructor initializes each data member to zero"""

      # each statement invokes __setattr__
      self.hour = hour
      self.minute = minute
      self.second = second
```

ATTRIBUTE ACCESS

```
def __setattr__( self, name, value ):
   """Assigns a value to an attribute"""

   if name == "hour":

      if 0 <= value < 24:
         self.__dict__[ "_hour" ] = value
      else:
         raise ValueError, "Invalid hour value: %d" % value

   elif name == "minute" or name == "second":

      if 0 <= value < 60:
         self.__dict__[ "_" + name ] = value
      else:
         raise ValueError, "Invalid %s value: %d" % \
            ( name, value )

   else:
      self.__dict__[ name ] = value
```

ATTRIBUTE ACCESS

```
def __getattr__( self, name ):
   """Performs lookup for unrecognized attribute name"""

   if name == "hour":
      return self._hour
   elif name == "minute":
      return self._minute
   elif name == "second":
      return self._second
   else:
      raise AttributeError, name
```

ATTRIBUTE ACCESS

```python
def __str__( self ):
    """Returns Time object string in military format"""

    # attribute access does not call __getattr__
    return "%.2d:%.2d:%.2d" % \
       ( self._hour, self._minute, self._second )
```

ATTRIBUTE ACCESS

```
>>> from TimeAccess import Time
>>> time1 = Time( 4, 27, 19 )
>>> print time1
04:27:19
>>> print time1.hour, time1.minute, time1.second
4 27 19
>>> time1.hour = 16
>>> print time1
16:27:19
>>> time1.second = 90
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "TimeAccess.py", line 30, in __setattr__
    raise ValueError, "Invalid %s value: %d" % \
ValueError: Invalid second value: 90
```

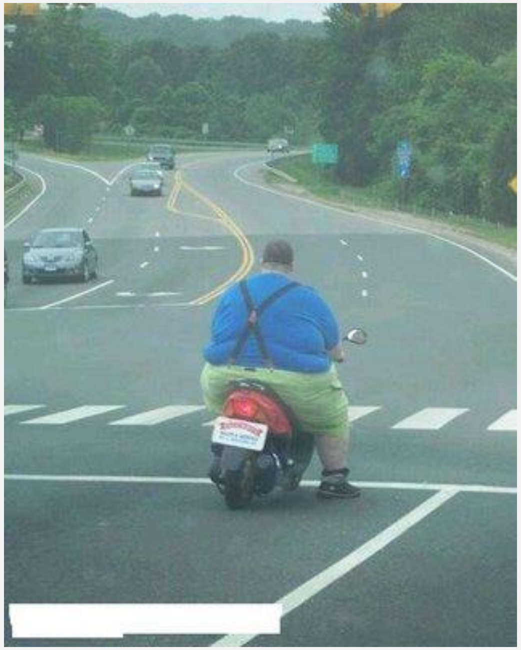
`time1.attribute` = `time1.__getattr__( attribute )`

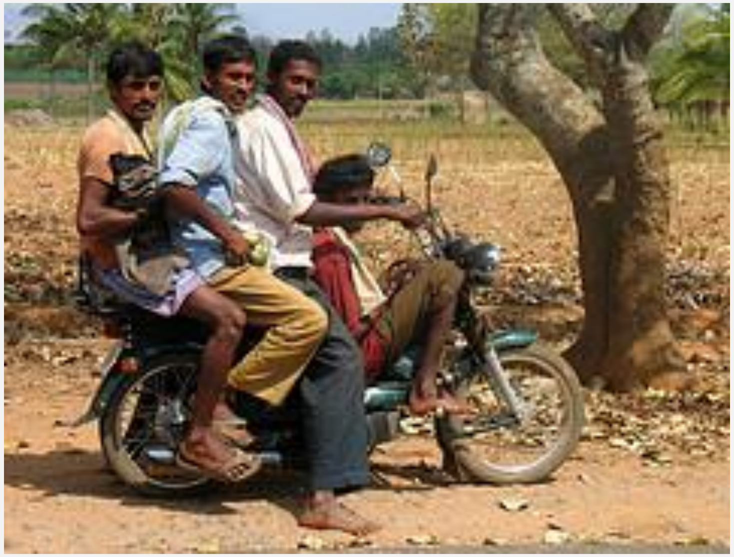
`self._hour = value` = `__setattr__`

OPERATOR OVERLOADING

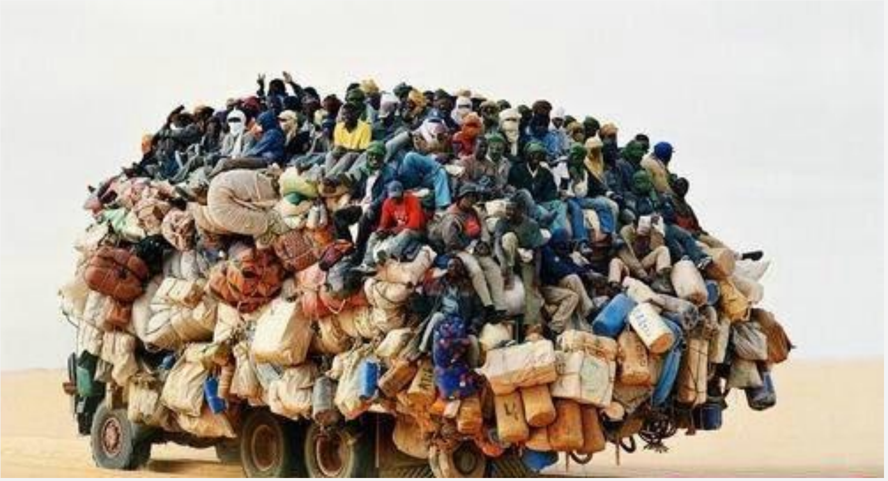
Python does not allow new operators to be created, it does allow most existing operators to be overloaded

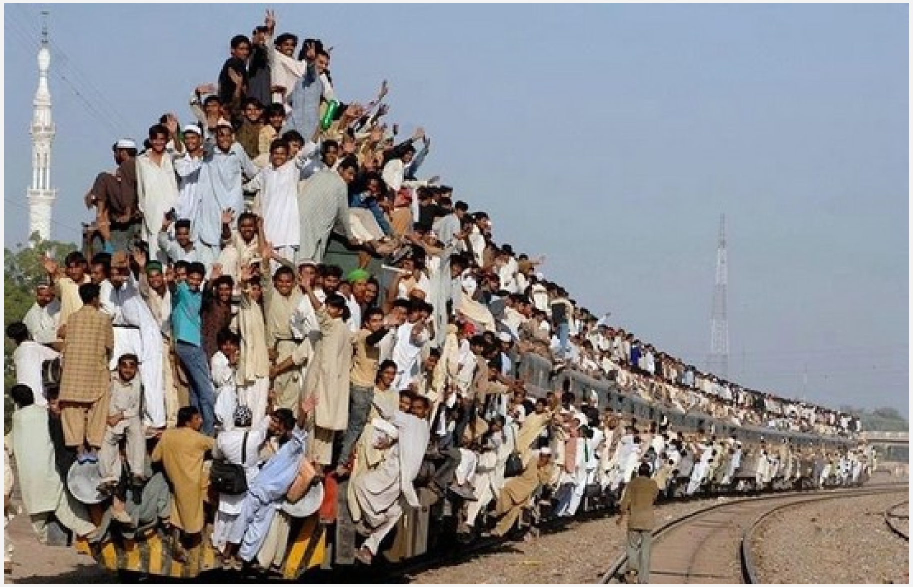
when these operators are used with objects of a programmer-defined type, the operators have meaning appropriate to the new types.





Overloading: It's about maximizing capabilities.





OPERATOR OVERLOADING

overloading contributes to the extensibility of Python language

OPERATOR OVERLOADING *Restrictions*

<table>
<tr><th colspan="8">Common operators and augmented assignment statements that can be overloaded</th></tr>
<tr><td>+</td><td>-</td><td>*</td><td>**</td><td>/</td><td>//</td><td>%</td><td><<</td></tr>
<tr><td>>></td><td>&</td><td>|</td><td>^</td><td>~</td><td><</td><td>></td><td><=</td></tr>
<tr><td>>=</td><td>==</td><td>!=</td><td>+=</td><td>-=</td><td>*=</td><td>**=</td><td>/=</td></tr>
<tr><td>//=</td><td>%=</td><td><<=</td><td>>>=</td><td>&=</td><td>^=</td><td>|=</td><td>[]</td></tr>
<tr><td>()</td><td>.</td><td>``</td><td>in</td><td></td><td></td><td></td><td></td></tr>
</table>

1. precedence cannot be changed
2. arity cannot be changed (i.e., unary to binary)

OPERATOR OVERLOADING *Restrictions*

<table>
<tr><th colspan="8">Common operators and augmented assignment statements that can be overloaded</th></tr>
<tr><td>+</td><td>-</td><td>*</td><td>**</td><td>/</td><td>//</td><td>%</td><td><<</td></tr>
<tr><td>>></td><td>&</td><td>|</td><td>^</td><td>~</td><td><</td><td>></td><td><=</td></tr>
<tr><td>>=</td><td>==</td><td>!=</td><td>+=</td><td>-=</td><td>*=</td><td>**=</td><td>/=</td></tr>
<tr><td>//=</td><td>%=</td><td><<=</td><td>>>=</td><td>&=</td><td>^=</td><td>|=</td><td>[]</td></tr>
<tr><td>()</td><td>.</td><td>``</td><td>in</td><td></td><td></td><td></td><td></td></tr>
</table>

```
object2 = object2 + object1
```

```
object2 += object1
```

OPERATOR OVERLOADING *Unary*

A unary operator for a class is overloaded as a method that takes only the object reference argument (self)

```
~object1
```

```
object1.__invert__()
```

OPERATOR OVERLOADING *Unary*

Unary operator	Special method
`-`	`__neg__`
`+`	`__pos__`
`~`	`__invert__`

OPERATOR OVERLOADING *Binary*

A binary operator or statement for a class is overloaded as a method with two arguments: self and other.

OPERATOR OVERLOADING *Rational*

A Rational number is a fraction represented as a numerator (top) and a denominator (bottom).

A rational number can be positive, negative or zero.

Class Rational's interface includes a default constructor, string representation method, overloaded **abs** function, equality operators and several mathematical operators.

OPERATOR OVERLOADING *Rational*

```
class Rational:
    """Representation of rational number"""

    def __init__( self, top = 1, bottom = 1 ):
        """Initializes Rational instance"""

        # do not allow 0 denominator
        if bottom == 0:
            raise ZeroDivisionError, "Cannot have 0 denominator"

        # assign attribute values
        self.numerator = abs( top )
        self.denominator = abs( bottom )
        self.sign = ( top * bottom ) / ( self.numerator *
            self.denominator )

        self.simplify()  # Rational represented in reduced form
```

OPERATOR OVERLOADING *Binary*

When overloading binary operator +, if y and z are objects of class Rational, then y + z is treated as if y.__add__(z) had been written, invoking the __add__ method.

OPERATOR OVERLOADING *Binary*

Usually, overloaded binary operator methods create and return new objects of their corresponding class.

OPERATOR OVERLOADING *Binary*

What happens if we evaluate the expression $y + z$ or the statement $y += z$, and only y is an object of class Rational?

In both cases, z must be coerced (i.e., converted) to an object of class Rational, before the appropriate operator overloading method executes.

OPERATOR OVERLOADING *Binary*

+	`__add__`, `__radd__`
-	`__sub__`, `__rsub__`
*	`__mul__`, `__rmul__`
/	`__div__`, `__rdiv__`, `__truediv__` (for Python 2.2), `__rtruediv__` (for Python 2.2)
//	`__floordiv__`, `__rfloordiv__` (for Python version 2.2)
%	`__mod__`, `__rmod__`
**	`__pow__`, `__rpow__`
<<	`__lshift__`, `__rlshift__`
>>	`__rshift__`, `__rrshift__`
&	`__and__`, `__rand__`
^	`__xor__`, `__rxor__`
|	`__or__`, `__ror__`

OPERATOR OVERLOADING *Binary*

`+=`	`__iadd__`
`-=`	`__isub__`
`*=`	`__imul__`
`/=`	`__idiv__`, `__itruediv__` (for Python version 2.2)
`//=`	`__ifloordiv__` (for Python version 2.2)
`%=`	`__imod__`
`**=`	`__ipow__`
`<<=`	`__ilshift__`
`>>=`	`__irshift__`
`&=`	`__iand__`
`^=`	`__ixor__`
`|=`	`__ior__`
`==`	`__eq__`
`!+, <>`	`__ne__`
`>`	`__gt__`
`<`	`__lt__`
`>=`	`__ge__`
`<=`	`__le__`

OPERATOR OVERLOADING *Built-ins*

A class also may define special methods that execute when certain built-in functions are called on an object of the class.

For example, we may define special method __abs__ for class Rational, to execute when a program calls abs(rationalObject) to compute the absolute value of an object of that class.

OPERATOR OVERLOADING *Built-ins*

Built-in Function	Description	Special method
`abs(` *x* `)`	Returns the absolute value of *x*.	`__abs__`
`divmod(` *x*, *y* `)`	Returns a tuple that contains the integer and remainder components of *x* `%` *y*.	`__divmod__`
`len(` *x* `)`	Returns the length of *x* (*x* should be a sequence).	`__len__`
`pow(` *x*, *y*[, *z*] `)`	Returns the result of x^y. With three arguments, returns (x^y) `%` *z*.	`__pow__`
`repr(` *x* `)`	Returns a formal string representation of x (i.e., a string from which object *x* can be replicated).	`__repr__`

TYPE CONVERSION

Sometimes all the operations "stay within a type."

For example, adding (concatenating) a string to a string produces a string. But, it is often necessary to convert or coerce data of one type to data of another type.

TYPE CONVERSION

Programmers can force conversions among built-in types by calling the appropriate Python function, such as int or float.

TYPE CONVERSION

But what about user-defined classes?

The interpreter cannot know how to convert among user-defined classes and built-in types.

TYPE CONVERSION

The programmer must specify how such conversions are to occur with special methods that override the appropriate Python functions.

TYPE CONVERSION

For example, a class can define special method __int__ that overloads the behavior of the call int(anObject) to return an integer representation of the object.

TYPE CONVERSION

Method	Description
`__coerce__`	Converts two values to the same type.
`__complex__`	Converts object to complex number type.
`__float__`	Converts object to floating-point number type.
`__hex__`	Converts object to hexidecimal string type.

TYPE CONVERSION

Method	Description
`__int__`	Converts object to integer number type.
`__long__`	Converts object to long integer number type.
`__oct__`	Converts object to octal string type.
`__str__`	Converts object to string type. Also used to obtain informal string representation of object (i.e., a string that simply describes object).

CASE STUDY: RATIONAL CLASS

Sample computations with Rational objects:

```
rational1: 1
rational2: 1/3
rational3: -1/2

1 / 1/3 = 3
-1/2 - 1/3 = -5/6
1/3 * -1/2 - 1 = -7/6

rational1 after adding rational2 * rational3: 5/6

5/6 <= 1/3 : 0
5/6 > -1/2 : 1

The absolute value of -1/2 is: 1/2

1/3 as an integer is: 0
1/3 as a float is: 0.333333333333
1/3 + 1 = 4/3
```

CASE STUDY: RATIONAL CLASS

```
def gcd( x, y ):
    """Computes greatest common divisor of two values"""

    while y:
        z = x
        x = y
        y = z % y
```

```
    return x
```

function **gcd()** computes the greatest common divisor of two values.

Class **Rational** uses this function to simplify the rational number.

CASE STUDY: RATIONAL CLASS

```
class Rational:
    """Representation of rational number"""

    def __init__( self, top = 1, bottom = 1 ):
        """Initializes Rational instance"""

        # do not allow 0 denominator
        if bottom == 0:
            raise ZeroDivisionError, "Cannot have 0 denominator"

        # assign attribute values
        self.numerator = abs( top )
        self.denominator = abs( bottom )
        self.sign = ( top * bottom ) / ( self.numerator *
            self.denominator )

        self.simplify()  # Rational represented in reduced form
```

CASE STUDY: RATIONAL CLASS

```python
# class interface method
def simplify( self ):
    """Simplifies a Rational number"""

    common = gcd( self.numerator, self.denominator )
    self.numerator /= common
    self.denominator /= common
```

CASE STUDY: RATIONAL CLASS

```
# overloaded binary arithmetic operators
def __add__( self, other ):
    """Overloaded addition operator"""

    return Rational(
        self.sign * self.numerator * other.denominator +
        other.sign * other.numerator * self.denominator,
        self.denominator * other.denominator )

def __sub__( self, other ):
    """Overloaded subtraction operator"""

    return self + ( -other )
```

CASE STUDY: RATIONAL CLASS

```
def __mul__( self, other ):
   """Overloaded multiplication operator"""

   return Rational( self.numerator * other.numerator,
                    self.sign * self.denominator *
                    other.sign * other.denominator )

def __div__( self, other ):
   """Overloaded / division operator."""

   return Rational( self.numerator * other.denominator,
                    self.sign * self.denominator *
                    other.sign * other.numerator )
```

CASE STUDY: RATIONAL CLASS

```
def __truediv__( self, other ):
    """Overloaded / division operator. (For use with Python
    versions (>= 2.2) that contain the // operator)"""

    return self.__div__( other )
```

CASE STUDY: RATIONAL CLASS

```
# overloaded binary comparison operators
def __eq__( self, other ):
    """Overloaded equality operator"""

    return ( self - other ).numerator == 0
```

```
def __ne__( self, other ):
    """Overloaded inequality operator"""

    return not ( self == other )
```

CASE STUDY: RATIONAL CLASS

```
def __lt__( self, other ):
    """Overloaded less-than operator"""

    return ( self - other ).sign < 0

def __gt__( self, other ):
    """Overloaded greater-than operator"""

    return ( self - other ).sign > 0

def __le__( self, other ):
    """Overloaded less-than or equal-to operator"""

    return ( self < other ) or ( self == other )

def __ge__( self, other ):
    """Overloaded greater-than or equal-to operator"""

    return ( self > other ) or ( self == other )
```

CASE STUDY: RATIONAL CLASS

```python
# overloaded built-in functions
def __abs__( self ):
   """Overloaded built-in function abs"""

   return Rational( self.numerator, self.denominator )
```

CASE STUDY: RATIONAL CLASS

```python
def __str__( self ):
    """String representation"""

    # determine sign display
    if self.sign == -1:
        signString = "-"
    else:
        signString = ""

    if self.numerator == 0:
        return "0"
    elif self.denominator == 1:
        return "%s%d" % ( signString, self.numerator )
    else:
        return "%s%d/%d" % \
            ( signString, self.numerator, self.denominator )
```

CASE STUDY: RATIONAL CLASS

```
# overloaded coercion capability
def __int__( self ):
    """Overloaded integer representation"""

    return self.sign * divmod( self.numerator,
       self.denominator )[ 0 ]

def __float__( self ):
    """Overloaded floating-point representation"""

   return self.sign * float( self.numerator ) / self.denominator
```

CASE STUDY: RATIONAL CLASS

```
def __coerce__( self, other ):
    """Overloaded coercion. Can only coerce int to Rational"""

    if type( other ) == type( 1 ):
        return ( self, Rational( other ) )
    else:
        return None
```

CASE STUDY: RATIONAL CLASS

```
from RationalNumber import Rational

# create objects of class Rational
rational1 = Rational()  # 1/1
rational2 = Rational( 10, 30 )  # 10/30 (reduces to 1/3)
rational3 = Rational( -7, 14 )  # -7/14 (reduces to -1/2)
```

```
# print objects of class Rational
print "rational1:", rational1
```

```
print "rational2:", rational2
print "rational3:", rational3
print
```

```
rational1: 1
rational2: 1/3
rational3: -1/2
```

CASE STUDY: RATIONAL CLASS

```
# test mathematical operators
print rational1, "/", rational2, "=", rational1 / rational2
print rational3, "-", rational2, "=", rational3 - rational2
print rational2, "*", rational3, "-", rational1, "=", \
   rational2 * rational3 - rational1
```

```
1 / 1/3 = 3
-1/2 - 1/3 = -5/6
1/3 * -1/2 - 1 = -7/6
```

CASE STUDY: RATIONAL CLASS

```
# overloading + implicitly overloads +=
rational1 += rational2 * rational3
print "\nrational1 after adding rational2 * rational3:", rational1
print
```

```
rational1 after adding rational2 * rational3: 5/6
```

CASE STUDY: RATIONAL CLASS

```
# test comparison operators
print rational1, "<=", rational2, ":", rational1 <= rational2
print rational1, ">", rational3, ":", rational1 > rational3
print
```

```
5/6 <= 1/3 : 0
5/6 > -1/2 : 1
```

CASE STUDY: RATIONAL CLASS

```
# test built-in function abs
print "The absolute value of", rational3, "is:", abs( rational3 )
print
```

```
The absolute value of -1/2 is: 1/2
```

CASE STUDY: RATIONAL CLASS

```python
# test coercion
print rational2, "as an integer is:", int( rational2 )
print rational2, "as a float is:", float( rational2 )
print rational2, "+ 1 =", rational2 + 1
```

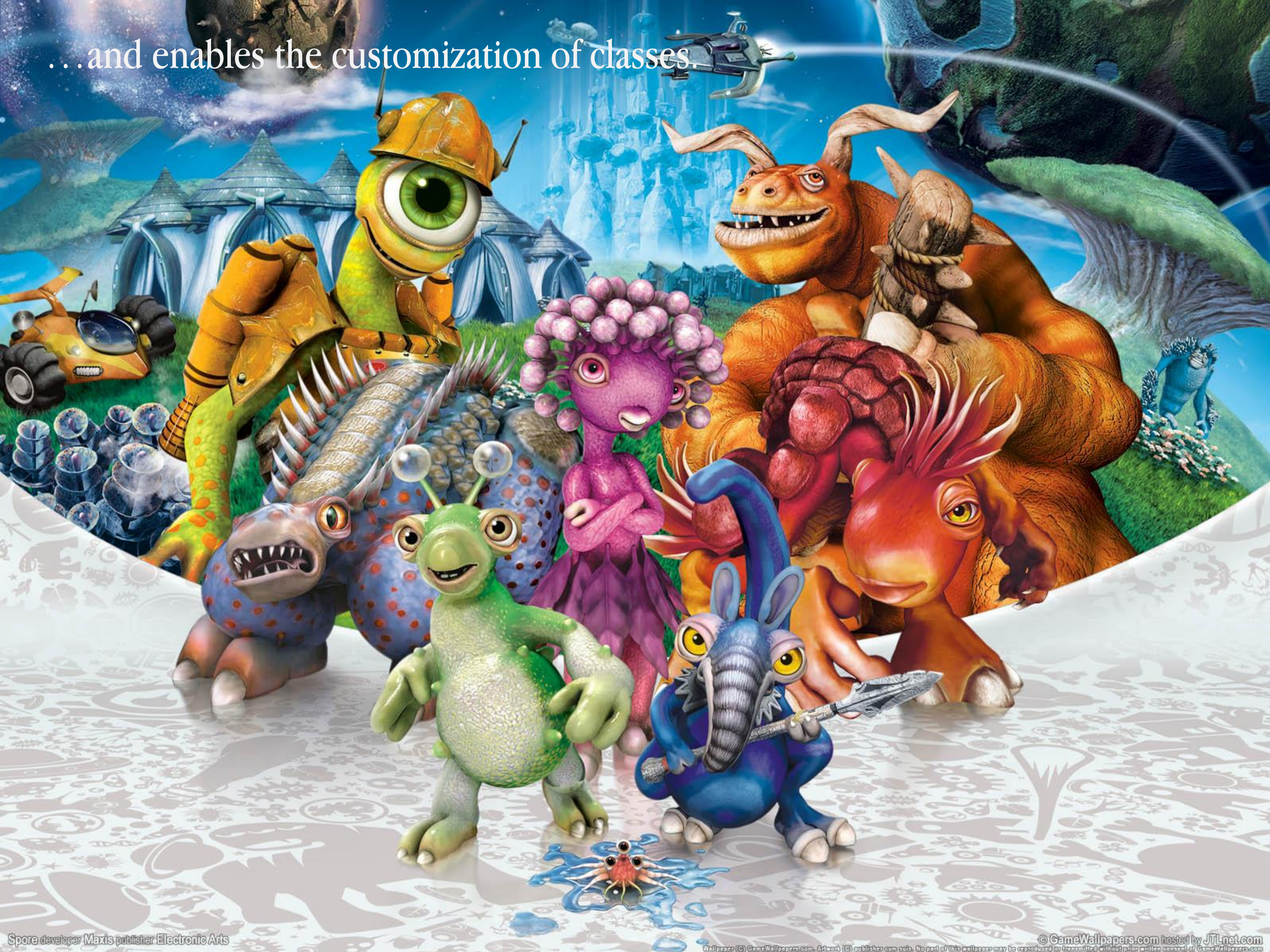
```
1/3 as an integer is: 0
1/3 as a float is: 0.333333333333
1/3 + 1 = 4/3
```

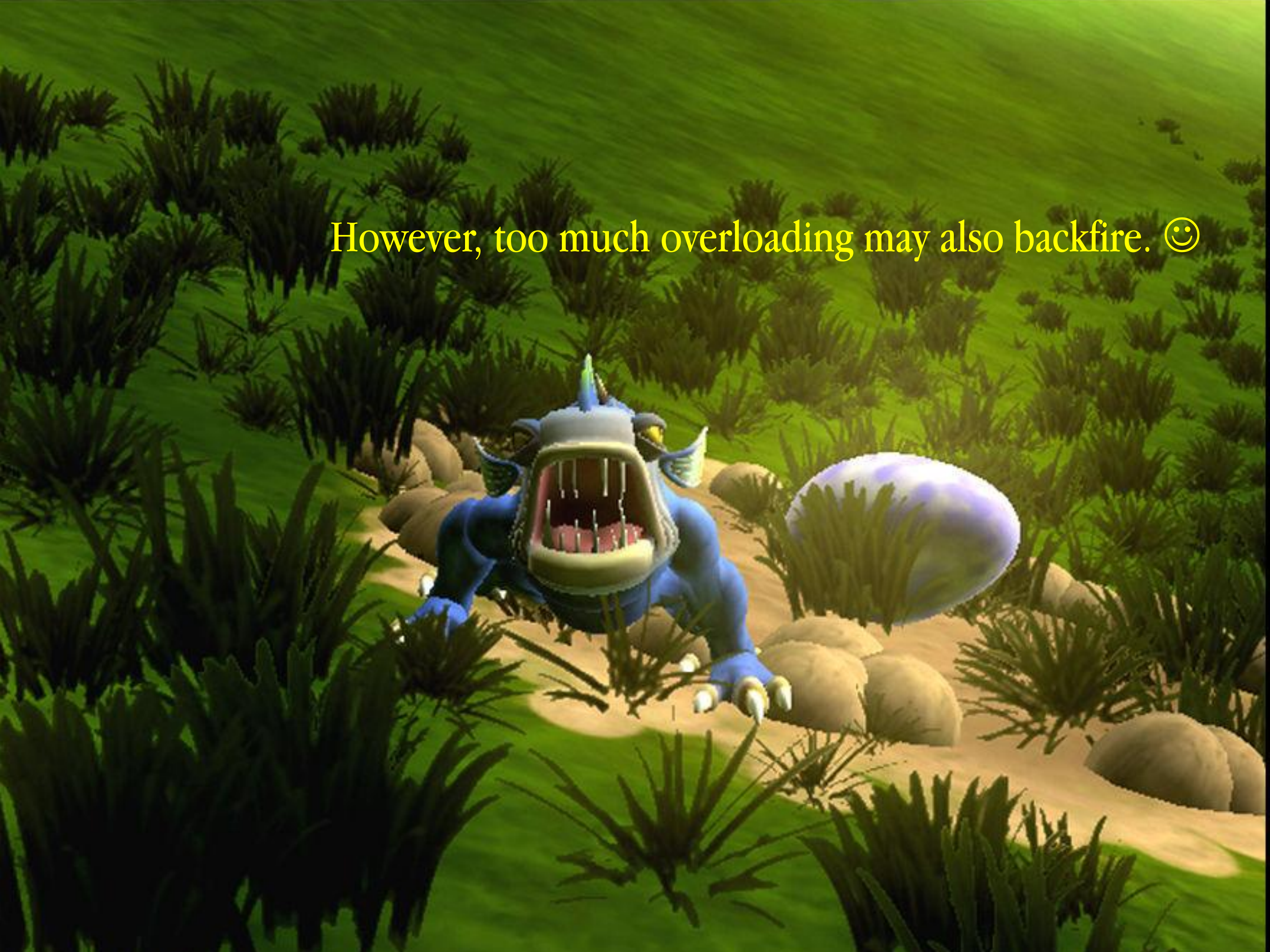
END NOTES

Overloading is a powerful concept.

It allows the programmer to reuse/modify existing concepts and operations.

…and enables the customization of classes.




However, too much overloading may also backfire. ☺

REFERENCES

- Deitel, Deitel, Liperi, and Wiedermann - Python: How to Program (2001).

- Disclaimer: Most of the images/information used here have no proper source citation, and I do not claim ownership of these either. I don't want to reinvent the wheel, and I just want to reuse and reintegrate materials that I think are useful or cool, then present them in another light, form, or perspective. Moreover, the images/information here are mainly used for illustration/educational purposes only, in the spirit of openness of data, spreading light, and empowering people with knowledge. ☺