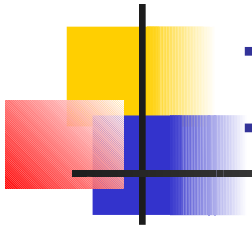
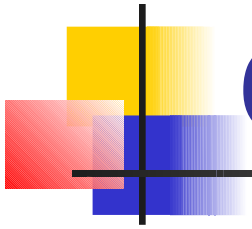


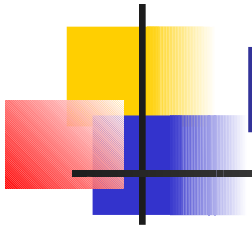
Introduction to OOP

- OOP = Object-Oriented Programming
- OOP is very simple in python
  - but also powerful
- What is an object?
  - data structure, and
  - functions (methods) that operate on it



OOP terminology

- class -- a template for building objects
- instance -- an object created from the template (an instance of the class)
- method -- a function that is part of the object and acts on instances directly
- constructor -- special "method" that creates new instances

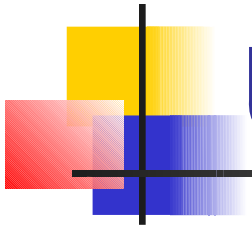


Defining a class

```
class Thingy:
    """This class stores an arbitrary object."""
    def __init__(self, value):
        """Initialize a Thingy."""
        self.value = value
    def showme(self):
        """Print this object to stdout."""
        print "value = %s" % self.value
```

constructor

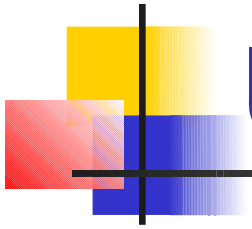
method



Using a class (1)

```
t = Thingy(10)  # calls __init__ method
t.showme()      # prints "value = 10"
```

- `t` is an instance of class `Thingy`
- `showme` is a method of class `Thingy`
- `__init__` is the constructor method of class `Thingy`
  - when a `Thingy` is created, the `__init__` method is called
- Methods starting and ending with `__` are "special" methods

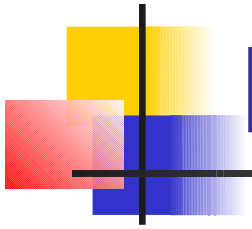


Using a class (2)

```
print t.value  # prints "10"
```

- **value** is a *field* of class **Thingy**

```
t.value = 20   # change the field value
print t.value  # prints "20"
```



More fun stuff

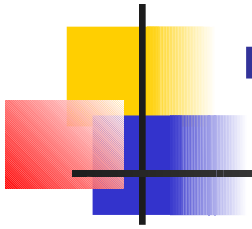
- Can write showme a different way:

```
def __repr__(self):
    return str(self.value)
```

- Now can do:

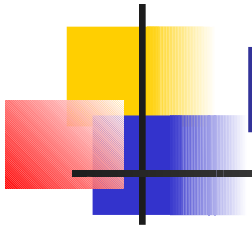
```
print t  # prints "10"
print "thingy: %s" % t  # prints "thingy: 10"
```

- `__repr__` converts object to string



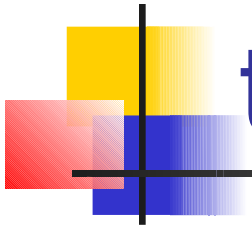
"Special" methods

- All start and end with `__` (two underscores)
- Most are used to emulate functionality of built-in types in user-defined classes
- *e.g.* operator overloading
  - `__add__`, `__sub__`, `__mult__`, `...`
  - see python docs for more information



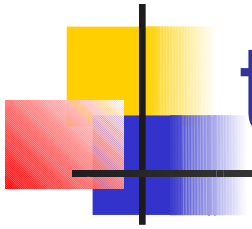
Exception handling

- What do we do when something goes wrong in code?
  - exit program (too drastic)
  - return an integer error code (clutters code)
- Exception handling is a cleaner way to deal with this
- Errors "raise" an exception
- Other code can "catch" an exception and deal with it



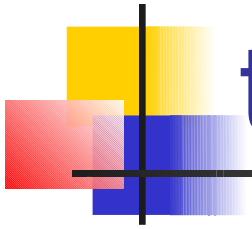
try/raise/except (1)

```
try:
    a = 1 / 0
    # this raises ZeroDivisionError
except ZeroDivisionError:
    # catch and handle the exception
    print "divide by zero"
    a = -1   # lame!
```



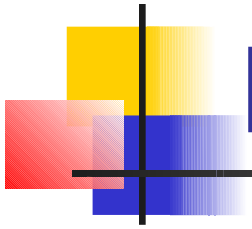
try/raise/except (2)

```
try:
    a = 1 / 0
    # this raises ZeroDivisionError
except:   # no exception specified
    # catches ANY exception
    print "something bad happened"
    # Don't do this!
```



try/raise/except (3)

```
try:
    a = 1 / 0
    # this raises ZeroDivisionError
except:   # no exception specified
    # Reraise original exception:
    raise
    # This is even worse!
```

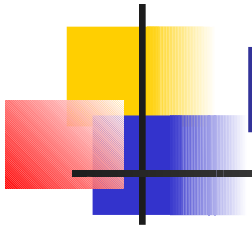


Backtraces

- Uncaught exceptions give rise to a stack backtrace:

```
# python bogus.py
Traceback (most recent call last):
  file "bogus.py", line 5, in ?
    foo()
  file "bogus.py", line 2, in foo
    a = 1 / 0
  ZeroDivisionError: integer division or modulo by
  zero
```

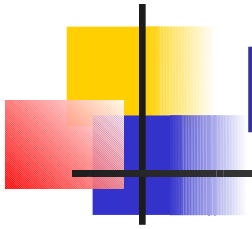
- Backtrace is better than catch-all exception handler



Exceptions are classes

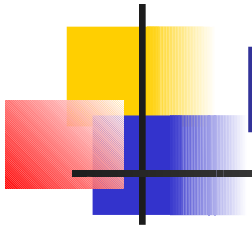
```
class SomeException:
    def __init__(self, value=None):
        self.value = value
    def __repr__(self):
        return `self.value`
```

- The expression `` `self.value` `` is the same as `str(value)`
- *i.e.* converts object to string



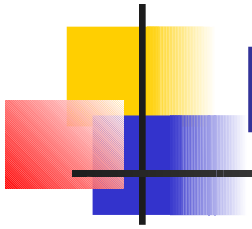
Raising exceptions (1)

```
def some_function():
    if something_bad_happens():
        # SomeException leaves function
        raise SomeException("bad!")
    else:
        # do the normal thing
```



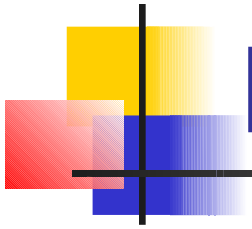
Raising exceptions (2)

```
def some_other_function():
    try:
        some_function()
    except SomeException, e:
        # e gets the exception that was caught
        print e.value
```



Raising exceptions (3)

```
# This is silly:
try:
    raise SomeException("bad!")
except SomeException, e:
    print e  # prints "bad!"
```



Random numbers (1)

- To use random numbers, import the **`random`** module; some useful functions include:

`random.choice(seq)`

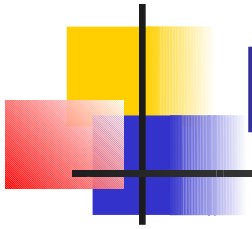
- chooses a random element from a sequence **`seq`** (usually a list)

`random.shuffle(seq)`

- randomizes the order of elements in a sequence **`seq`** (usually a list)

`random.sample(seq, k)`

- chooses k random elements from **`seq`**



Random numbers (2)

- To use random numbers, import the **`random`** module; some useful functions include:

`random.randrange(start, stop)`

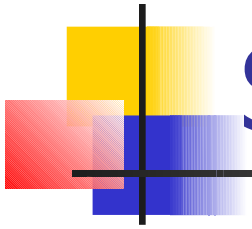
- chooses a random element from the range [**`start`**, **`stop`**] (not including the endpoint)

`random.randint(start, stop)`

- chooses a random element from the range [**`start`**, **`stop`**] (including the endpoint)

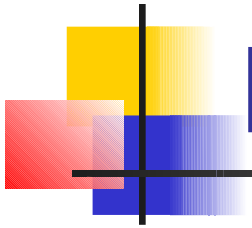
`random.random()`

- returns a random float in the range (**`0`**, **`1`**)



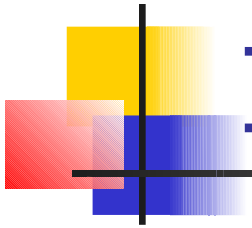
Summing up

- Use classes where possible
- Use exceptions to deal with error situations
- Use docstrings for documentation
- Next week: more OOP (inheritance)



More on OOP -- inheritance

- Often want to create a class which is a specialization of a previously-existing class
- Don't want to redefine the entire class from scratch
  - Just want to add a few new methods and fields
- To do this, the new class can inherit from another class; this is called inheritance
- The class being inherited from is called the parent class, base class, or superclass
- The class inheriting is called the child class, derived class, or subclass



Inheritance (2)

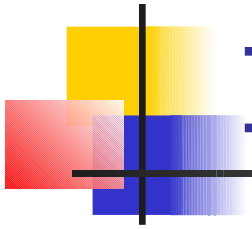
- Inheritance:

```
class DerivedClass(BaseClass):
    <statement-1>
    ...
    <statement-N>
```

- Or:

```
class DerivedClass(mod.BaseClass):
    # ...
```

  - if BaseClass is defined in another module ("**mod**")

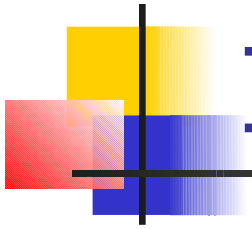


Inheritance (3)

- Name resolution:

```
foo = Foo() # instance of class Foo
foo.bar()
```

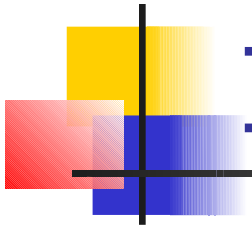
- If **`bar`** method not in class **`Foo`**
  - parent class of **`Foo`** searched
  - etc. until **`bar`** found or top reached
  - **`AttributeError`** raised if not found
  - Same deal with fields (**`foo.x`**)



Inheritance (4)

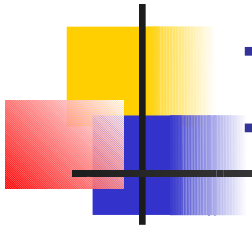
- Constructors:
  - Calling `__init__` method on subclass doesn't automatically call superclass constructor!
  - Can call superclass constructor explicitly if necessary

Inheritance (5)



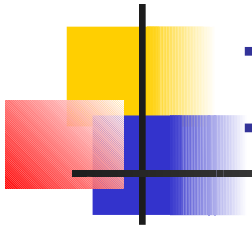
```
class base:
  def __init__(self, x):
    self.x = x

class derive (base):
  def __init__(self, y):
    base.__init__(self, y)
    self.y = y
```



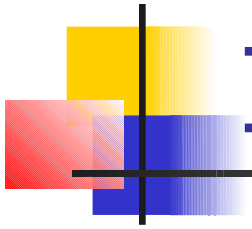
Inheritance example (1)

```
class Animal:
    def __init__(self, weight):
        self.weight = weight
    def eat(self):
        print "I am eating!"
    def __repr__(self):
        return "Animal; weight = %d" % \
            self.weight
```



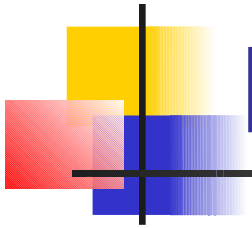
Inheritance example (2)

```
>>> a = Animal(100)
>>> a.eat()
I am eating!
>>> a.weight
100
>>> a.fly()
AttributeError: Animal instance has no
  attribute 'fly'
```



Inheritance example (3)

```
class Bird(Animal):
    def fly(self):
        print "I am flying!"
b = Bird(100) # Animal's __init__() method
b.eat()
I am eating!
b.fly()
I am flying!
```

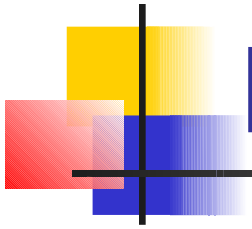


Multiple inheritance (1)

- Multiple inheritance:

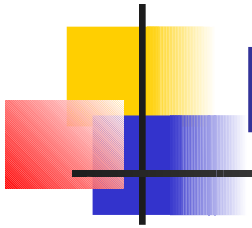
```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1> . .
    <statement-N>
```

- Resolution rule for repeated attributes:
- Left-to-right, depth first search
  - sorta...
  - Actual rules are slightly more complex
  - Don't depend on this if at all possible!



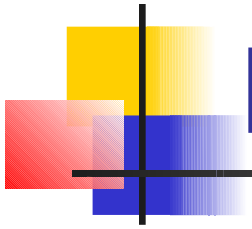
Multiple inheritance (2)

- Detailed rules:
  - **http://www.python.org/2.3/mro.html**
- Usually used with "mixin" classes
  - Combining two completely independent classes
  - Ideally no fields or methods shared
  - Conflicts then do not arise



Mixin example

```
class DNASequence:
    # __init__ etc.
    def getBaseCounts(self): # ...
    # other DNA-specific methods
class DBStorable:
    # __init__ etc.
    # methods for storing into database
class StorableDNASequence(DNASequence, \
    DBStorable):
    # Override methods as needed
    # No common fields/methods in superclasses
```



Private fields

- Private fields of objects
  - at least two leading underscores
  - at most one trailing underscore
  - *e.g.* **`__spam`**

- **`__spam`** → **`_<classname>__spam`**
  - **`<classname>`** is current class name
- Weak form of privacy protection