Week 10

Object-oriented programming

Writing Games with Pygame

Object-oriented programming

- Object-oriented programming (OOP) is a way of programming (paradigm) that allows mechanisms used with real-life entities to be brought to code.

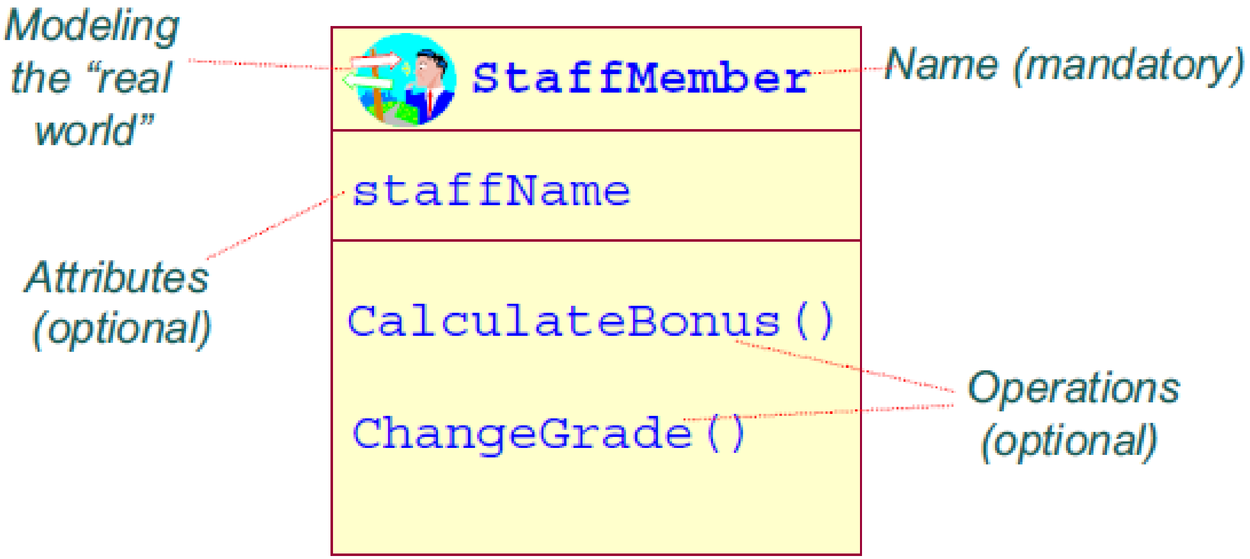
Benefits

- **Encapsulation:** Allows the code to be packaged within a unit (object) where the scope of action can be determined.
- **Abstraction:** It allows to generalize the types of objects through the classes and to simplify the program.
- **Inheritance:** Allows code reuse by being able to inherit attributes and behaviors from one class to another.
- **Polymorphism:** Allows you to create multiple objects from the same flexible piece of code.

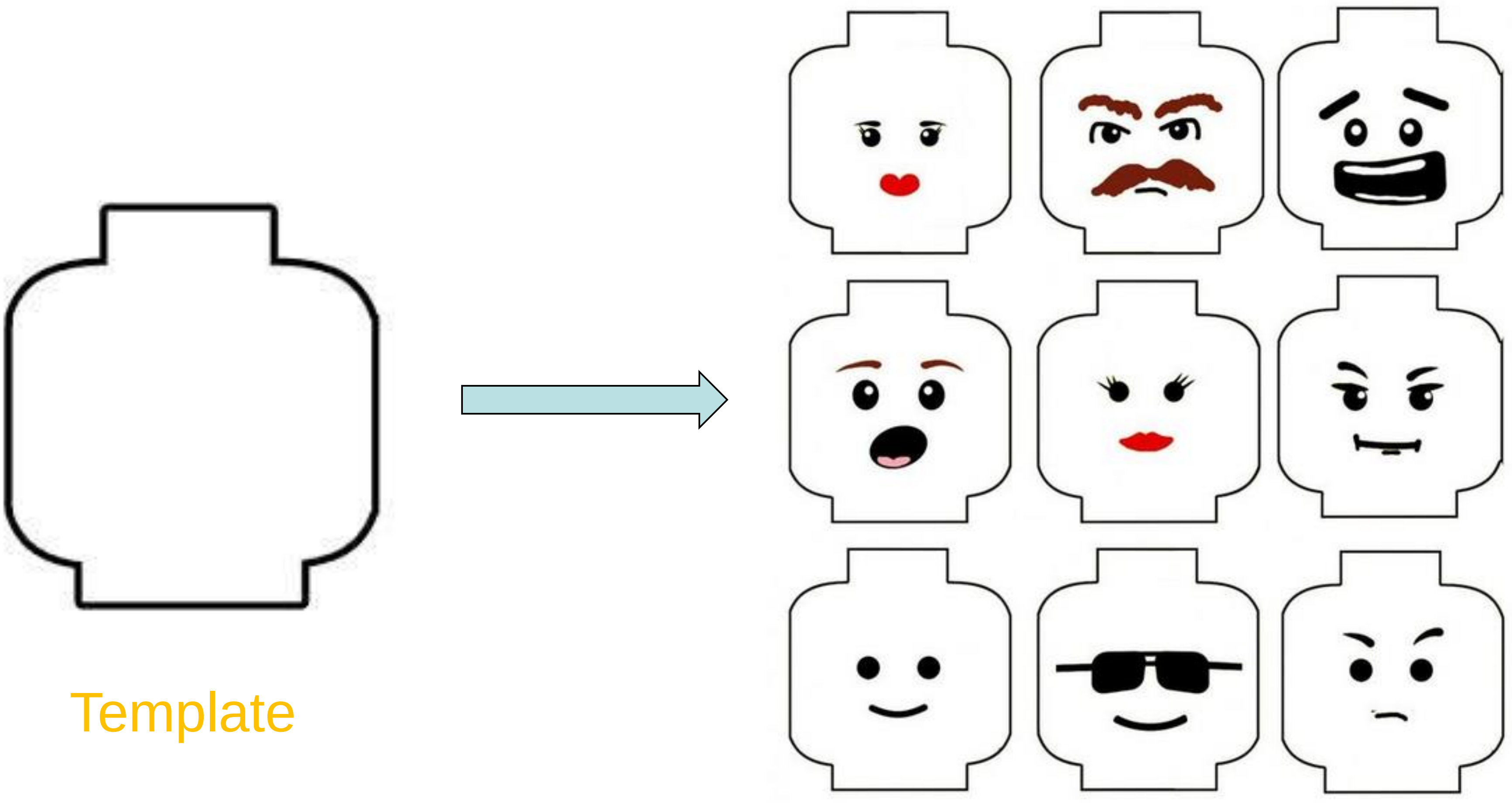
Classes

- *A class describes a group of objects with*
  - similar properties (attributes),
  - common behavior (operations),
  - common relationships with other objects
  - and common meaning (“semantics”).
- Finding classes: Listen to the domain experts (…the people who know the domain you are modeling!)

Diagrammatic Notation for Classes



Objects are Class Instances

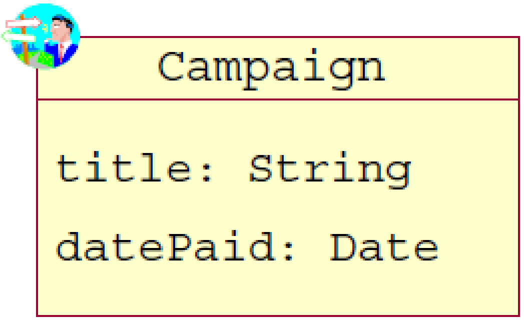


Template

Instances

Attributes: Class-object

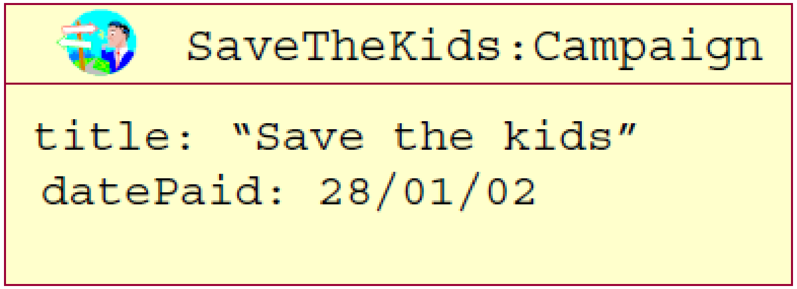
- *Each class can have **attttributes** which represent useful information about instances of a class.*
- *For example, Campaign has attributes title and datePaid.*



Campaign

title: String

datePaid: Date

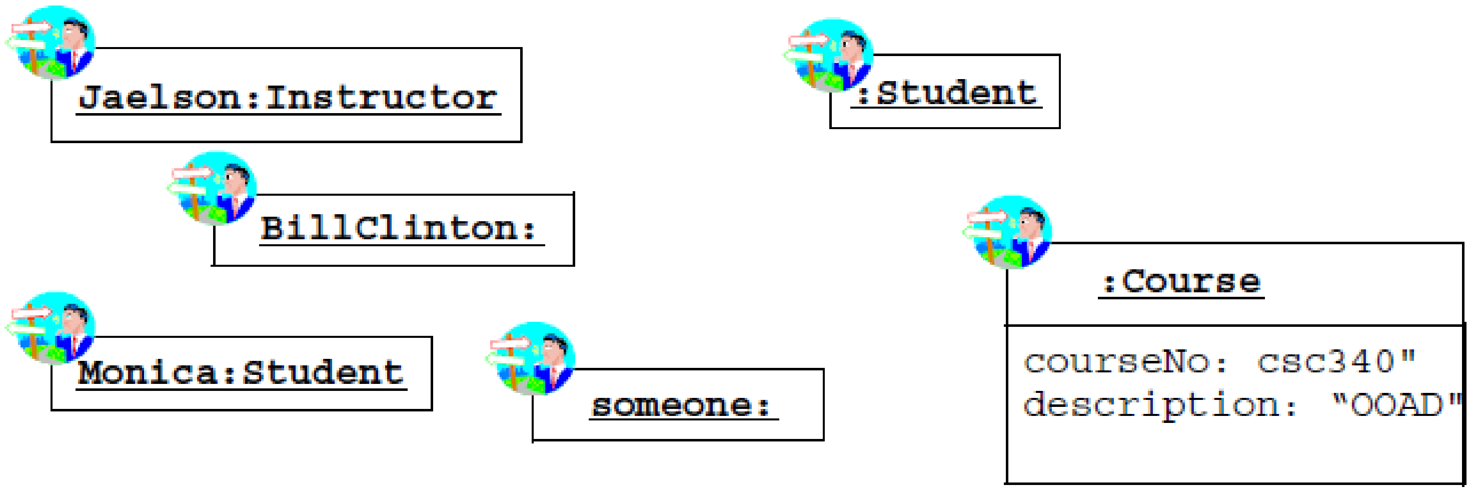


SaveTheKids:Campaign

title: “Save the kids”

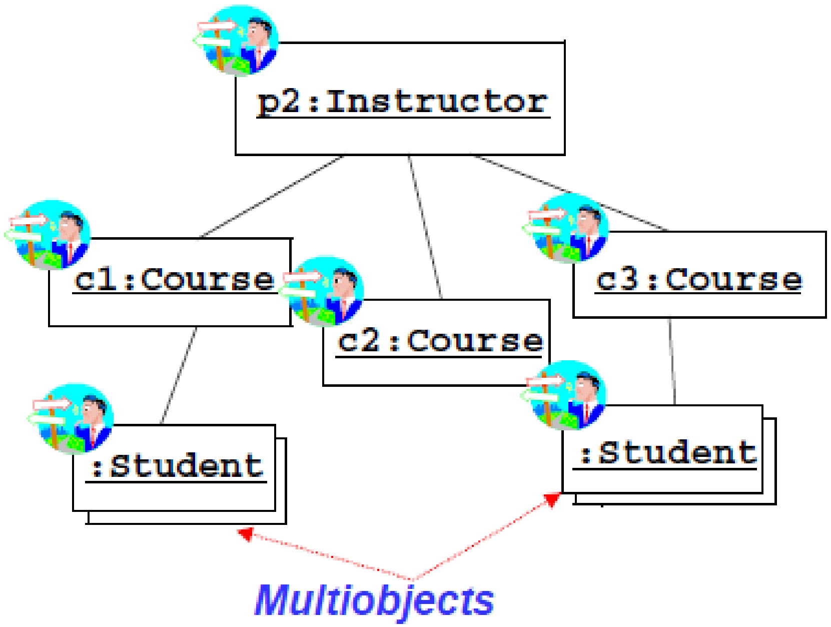
datePaid: 28/01/02

Object Diagrams



Multiobjects

- A multiobject is a set of objects, with an undefined
- number of elements



Terminology: Method/ Operations

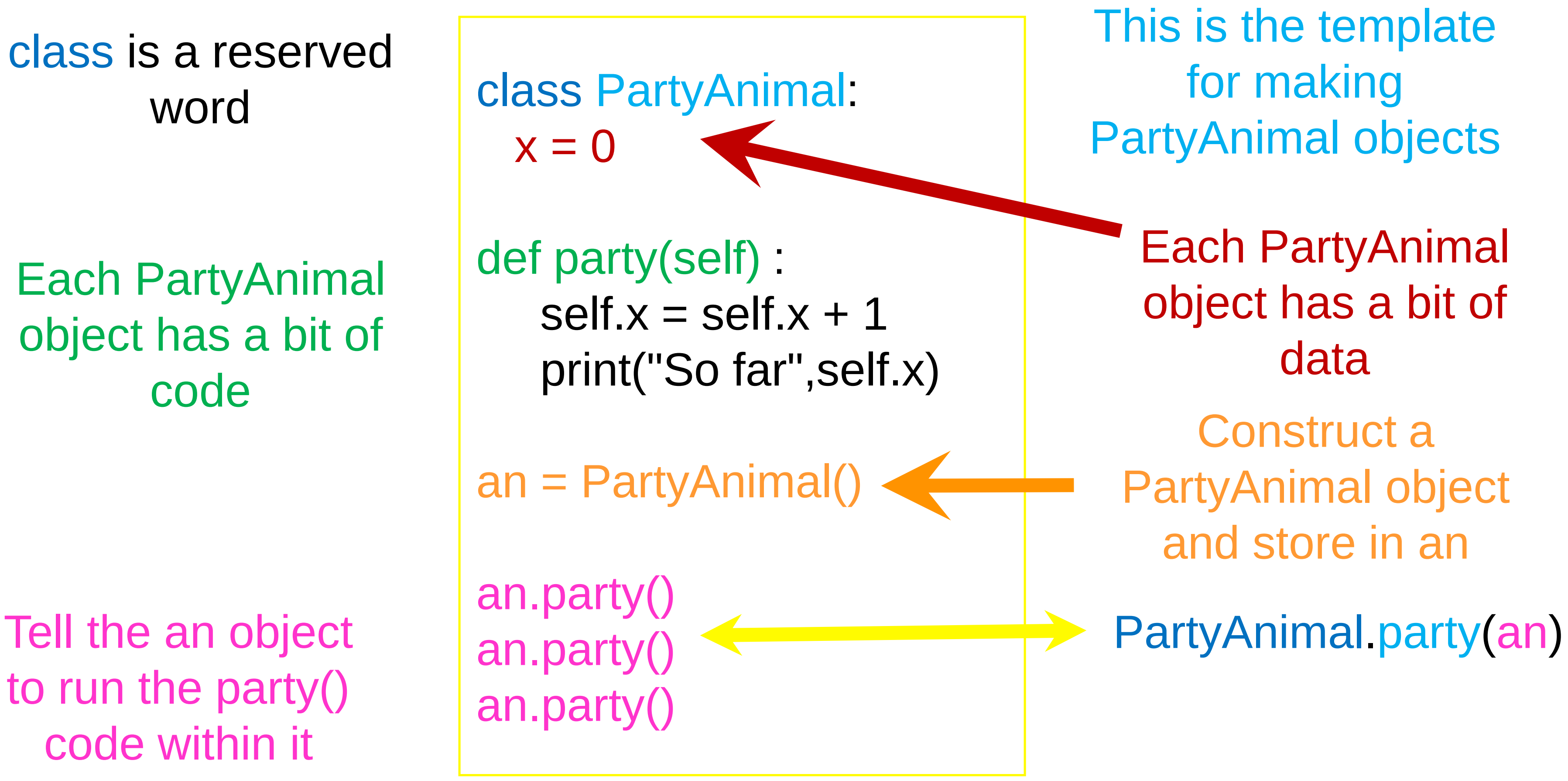
- Often derived from action verbs in the description of the Application
- Operations describe what can be done with the instances of a class.

Some Python Objects

```
>>> x = 'abc'
>>> type(x)
<class 'str'>
>>> type(2.5)
<class 'float'>
>>> type(2)
<class 'int'>
>>> y = list()
>>> type(y)
<class 'list'>
>>> z = dict()
>>> type(z)
<class 'dict'>
```

```
>>> dir(x)
[ … 'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', … 'lower',
'lstrip', 'maketrans', 'partition',
'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper',
'zfill']
>>> dir(y)
[… 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>> dir(z)
[…, 'clear', 'copy', 'fromkeys', 'get',
'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
```

Sample



class is a reserved word

```
class PartyAnimal:
   x = 0

def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

This is the template for making PartyAnimal objects

Each PartyAnimal object has a bit of data

Each PartyAnimal object has a bit of code

Construct a PartyAnimal object and store in an

Tell the an object to run the party() code within it

PartyAnimal.party(an)

Run

```
class PartyAnimal:
  x = 0

  def party(self) :
    self.x = self.x + 1
    print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

```
$ python party1.py
```

Run

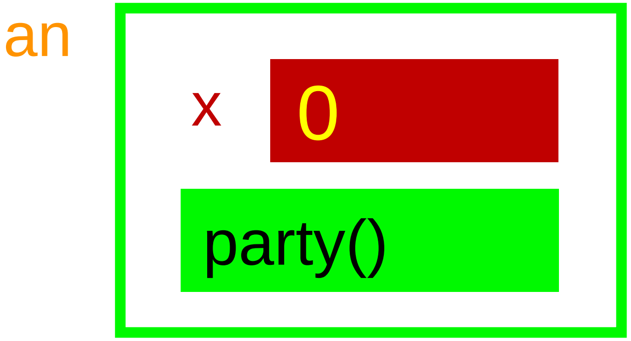
```
class PartyAnimal:
  x = 0

  def party(self) :
    self.x = self.x + 1
    print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

```
$ python party1.py
```



an

x 0

party()

Run

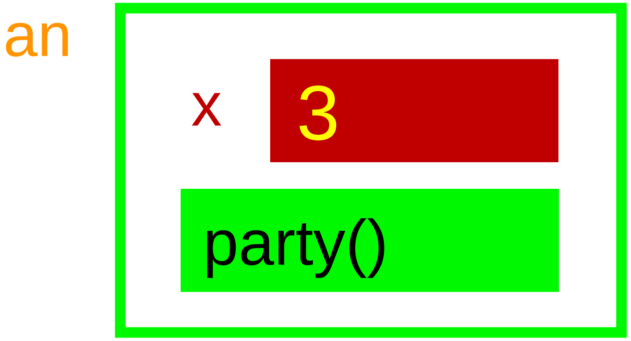
```
class PartyAnimal:
  x = 0

  def party(self) :
    self.x = self.x + 1
    print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

```
$ python party1.py
So far 1
So far 2
So far 3
```



PartyAnimal.party(an)

A Way to Find Capabilities

The dir() command lists capabilities

Ignore the ones with underscores - **these are used by Python itself**

The rest are real operations that the object can perform

It is like type() - it tells us something *about* a variable

```
>>> y = list()
>>> type(y)
<class 'list'>
>>> dir(y)
    ['__add__', '__class__',
             '__contains__',
              '__delattr__',
              '__delitem__',
  '__delslice__', '__doc__',
            … '__setitem__',
  '__setslice__', '__str__',
  'append', 'clear', 'copy',
 'count', 'extend', 'index',
  'insert', 'pop', 'remove',
          'reverse', 'sort']
>>>
```

A Way to Find Capabilities

```
class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()

print("Type", type(an))
print("Dir ", dir(an))
```

We can use dir() to find the “capabilities” of our newly created class.

```
$ python party3.py
  Type <class '__main__.PartyAnimal'>
 Dir  ['__class__', ... 'party', 'x']
```

Try dir() with a String

```
>>> x = 'Hello there'
>>> dir(x)
???
```

Object Lifecycle

- Objects are created, used, and discarded
- We have special blocks of code (methods) that get called
  - At the moment of creation (constructor)
  - At the moment of destruction (destructor)
- Constructors are used a lot
- Destructors are seldom used

Constructor

- The primary purpose of the constructor is to set up some instance variables to have the proper initial values when the object is created
- In object oriented programming, a constructor in a class is a special block of statements called when an object is created

Object Lifecycle sample

```
$ python party4.py
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

The constructor and destructor are optional. The constructor is typically used to set up variables. The destructor is seldom used.

```
class PartyAnimal:
   x = 0

   def __init__(self):
     print('I am
constructed')

   def party(self) :
     self.x = self.x + 1
     print('So far',self.x)

   def __del__(self):
     print('I am destructed',
self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```

Object Lifecycle

- Objects are created, used, and discarded
- We have special blocks of code (methods) that get called
  - At the moment of creation (constructor)
  - At the moment of destruction (destructor)
- Constructors are used a lot
- Destructors are seldom used

Many Instances

- We can create lots of objects - the class is the template for the object
- We can store each distinct object in its own variable
- We call this having multiple instances of the same class
- Each instance has its own copy of the instance variables

Many Instances

```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, z):
     self.name = z
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party
count",self.x)

s = PartyAnimal("Sally")
j = PartyAnimal("Jim")

s.party()
j.party()
s.party()
```

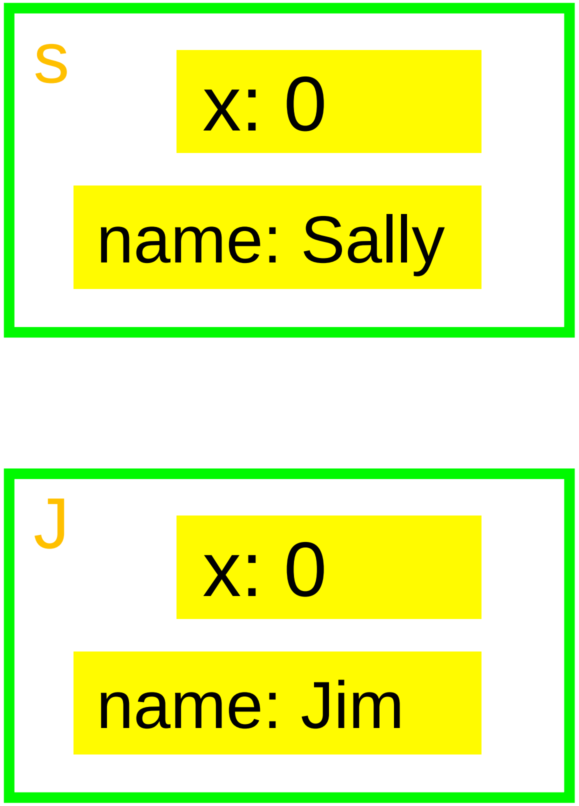
- Constructors can have additional parameters.
- These can be used to set up instance variables for the particular instance of the class (i.e., for the particular object).

Many Instances

```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, z):
     self.name = z
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party
count",self.x)

s = PartyAnimal("Sally")
j = PartyAnimal("Jim")
```



s

x: 0

name: Sally

J

x: 0

name: Jim

We have two independent instances

Many Instances

```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, z):
     self.name = z
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party
count",self.x)

s = PartyAnimal("Sally")
j = PartyAnimal("Jim")

s.party()
j.party()
s.party()
```

```
Sally constructed
Jim constructed
Sally party count 1
Jim party count 1
Sally party count 2
```

Inheritance

- When we make a new class - we can reuse an existing class and inherit all the capabilities of an existing class and then add our own little bit to make our new class
- Another form of store and reuse
- Write once - reuse many times
- The new class (child) has all the capabilities of the old class (parent) - and then some more

Inheritance

- ‘Subclasses’ are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own.

Inheritance

```
class name(superclass):
    statements
```

- Example:

```
class Point3D(Point):    # Point3D extends Point
    z = 0                # add a z field
    ...
```

- Python also supports *multiple inheritance*

```
class name(superclass, ..., superclass):
    statements
```

Calling Superclass Methods

- methods: **class** . **method** (**parameters**)
- constructors: **class** . __init__ (**parameters**)

```
class Point3D(Point):
    z = 0

    def __init__(self, x, y, z):
        Point.__init__(self, x, y)
        self.z = z

    def translate(self, dx, dy, dz):
        Point.translate(self, dx, dy)
        self.z += dz
```

Example

```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, nam):
     self.name = nam
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party count",self.x)

class FootballFan(PartyAnimal):
   points = 0
   def touchdown(self):
      self.points = self.points + 7
      self.party()
      print(self.name,"points",self.points)
```

```
s = PartyAnimal("Sally")
s.party()

j = FootballFan("Jim")
j.party()
j.touchdown()
```

FootballFan is a class which extends PartyAnimal. It has all the capabilities of PartyAnimal and more.

Example

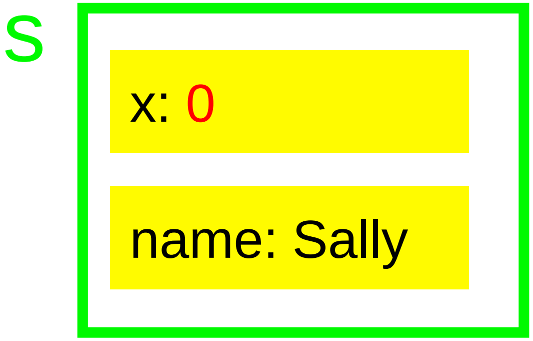
```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, nam):
     self.name = nam
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party count",self.x)

class FootballFan(PartyAnimal):
   points = 0
   def touchdown(self):
      self.points = self.points + 7
      self.party()
      print(self.name,"points",self.points)
```

```
s = PartyAnimal("Sally")
s.party()

j = FootballFan("Jim")
j.party()
j.touchdown()
```



s

x: 0

name: Sally

Example

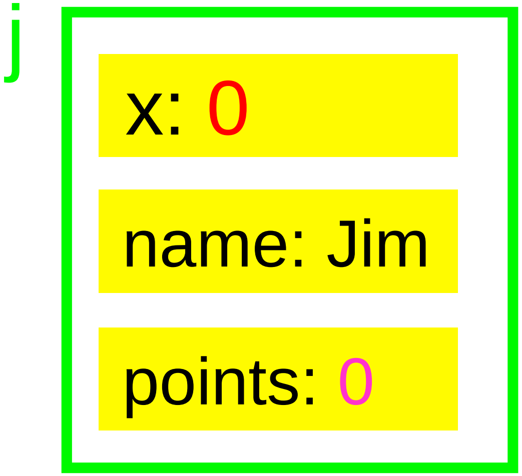
```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, nam):
     self.name = nam
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party count",self.x)

class FootballFan(PartyAnimal):
   points = 0
   def touchdown(self):
      self.points = self.points + 7
      self.party()
      print(self.name,"points",self.points)
```

```
s = PartyAnimal("Sally")
s.party()

j = FootballFan("Jim")
j.party()
j.touchdown()
```

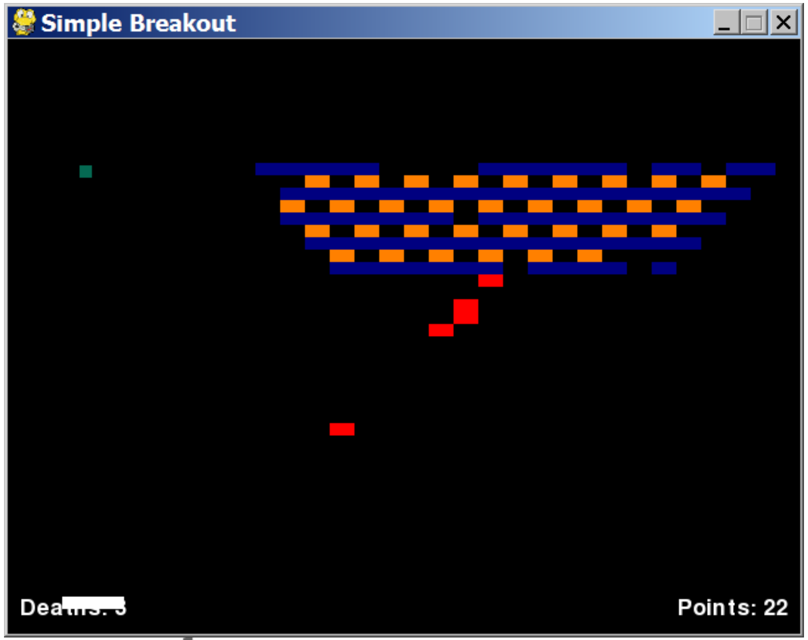


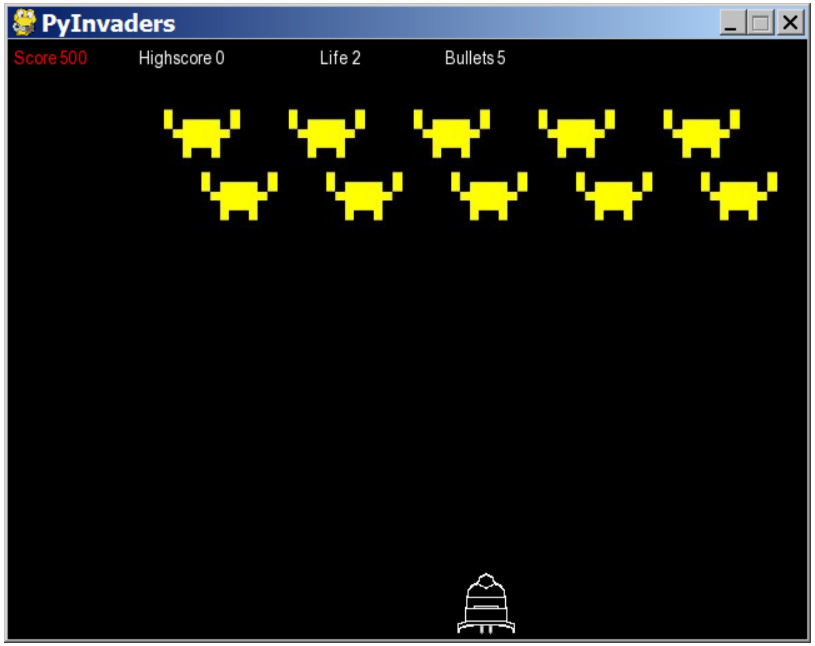
Pygame

Takes the "C++" out of "Game Development"

An Open Source Community Project

- A set of Python modules to help write games
- Deals with media (pictures, sound) nicely
- Interacts with user nicely (keyboard, joystick, mouse input)





Installing Pygame

- Go to the Pygame web site: http://www.pygame.org/
  - click 'Downloads' at left
  - Windows users: under the 'Windows' section,
    - click the most recent version (as of this quarter, that is pygame-1.9.1.win32-py2.6.msi)
  - Mac users: under the 'Macintosh' section,
    - click the most recent version (as of this quarter, pygame-1.9.1release-py2.6-macosx10.5.zip)
  - save file to hard disk
  - run file to install it

Other Resources

- Pygame documentation: http://www.pygame.org/docs/
  - lists every class in Pygame and its useful behavior

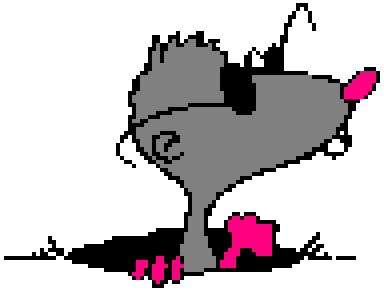
- The Application Programming Interface (API)
  - specifies the classes and functions in package

- Search for tutorials

- Experiment!

Our Goal: Whack-a-Mole

- Clicking on the mole plays a sound and makes mole move
- Number of hits is displayed at top of screen
- Enhancements
  - hit the mole with a shovel cursor
  - make the mole move around every 1 second if he's not hit



Initializing a Game

- Import Pygame's relevant classes:

```
import sys
import pygame
from pygame import *
from pygame.locals import *
from pygame.sprite import *
```

- Initialize Pygame at the start of your code:

```
pygame.init()
```

Creating a Window

name `= display.set_mode((`**width**`, `**height**`)` **[, options]**`)`

Example:

```
screen = display.set_mode((640, 480))
```

- Options:

`FULLSCREEN`	- use whole screen instead of a window
`DOUBLEBUF`	- display buffering for smoother animation
`OPENGL`	- 3D acceleration (don't use unless needed)

Example:

```
screen = display.set_mode((1024, 768), FULLSCREEN)
```

Initial Game Program

- An initial, incomplete game file using Pygame:

whack_a_mole.py

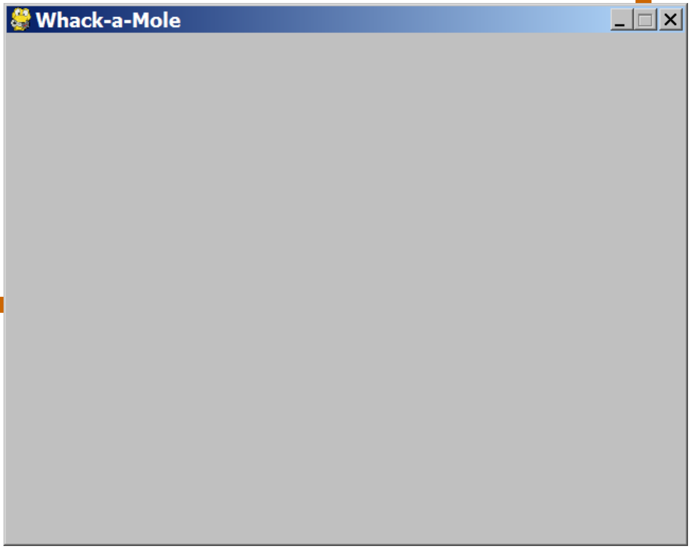
```
import pygame
from pygame import *
from pygame.locals import *
from pygame.sprite import *

pygame.init()

# set window title
display.set_caption("Whack-a-Mole")

screen = display.set_mode((640, 480))

```

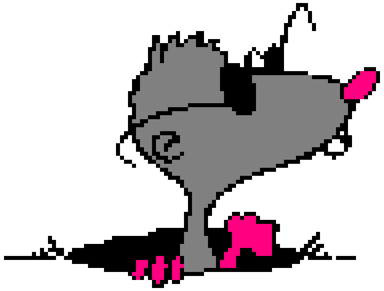


Sprites

Next we must define all the *sprites* found in the game.

- **sprite**: A character, enemy, or other object in a game.
  - Sprites can move, animate, collide, and be acted upon
  - Sprites usually consist of an *image* to draw on the screen and a *bounding rectangle* indicating the sprite's collision area

- Pygame sprites are objects that extend the `Sprite` class.



Programming a Sprite

```
class name(Sprite):
    # constructor
    def __init__(self):
        Sprite.__init__(self)
        self.image = image.load("filename")
        self.rect = self.image.get_rect()
```

other methods (if any)

- Pre-defined fields in every sprite:

  `self.image` - the image or shape to draw for this sprite

  - images are `Surface` objects, loaded by `image.load` function

  `self.rect` - position and size of where to draw the image

Sprite Example

```
# A class for a mole sprite to be whacked.
class Mole(Sprite):
    def __init__(self):
        Sprite.__init__(self)
        self.image = image.load("mole.gif")
        self.rect = self.image.get_rect()
```

Sprite Groups

name = `Group(`**sprite1**`,` **sprite2**`,` **...**`)`

- To draw sprites on screen, they must be put into a `Group`

Example:

```
my_mole = Mole()       # create a Mole object
all_sprites = Group(my_mole)
```

`Group` methods:

– `draw(`**surface**`)`	- draws all sprites in group onto a surface
– `update()`	- updates every sprite's appearance

Surface

- In Pygame, every 2D object is an object of type `Surface`
  - The screen object returned from `display.set_mode()`, each game character, images, etc.
  - Useful methods in each `Surface` object:

Method Name	Description
`fill((`**red**`, `**green**`, `**blue**`))`	paints surface in given color *(rgb 0-255)*
`get_width()`, `get_height()`	returns the dimensions of the surface
`get_rect()`	returns a `Rect` object representing the x/y/w/h bounding this surface
`blit(`**src**`, `**dest**`)`	draws this surface onto another surface

Drawing and Updating

- All Surface and Group objects have an `update` method that redraws that object when it moves or changes.

- Once sprites are drawn onto the screen, you must call `display.update()` to see the changes

```
my_mole = Mole()      # create a Mole object
all_sprites = Group(my_mole)
all_sprites.draw(screen)
display.update()      # redraw to see the sprites
```

Game Program v2

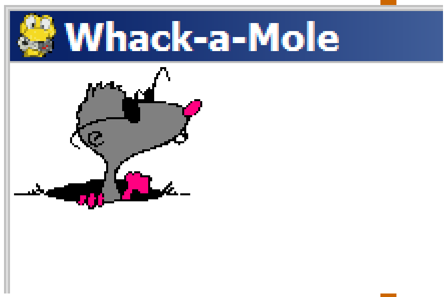
whack_a_mole.py

```
import pygame
from pygame import *
from pygame.locals import *
from pygame.sprite import *

class Mole(Sprite):
    def __init__(self):
        Sprite.__init__(self)
        self.image = image.load("mole.gif")
        self.rect = self.image.get_rect()

# main
pygame.init()
display.set_caption("Whack-a-Mole")
screen = display.set_mode((640, 480))

my_mole = Mole()                  # initialize sprites
all_sprites = Group(my_mole)
screen.fill((255, 255, 255))      # white background
all_sprites.draw(screen)
display.update()
```



Event-Driven Programming

- **event**: A user interaction with the game, such as a mouse click, key press, clock tick, etc.

- **event-driven programming**: Programs with an interface that waits for user events and responds to those events.

- Pygame programs need to write an *event loop* that waits for a Pygame event and then processes it.

Event Loop Template

```
# after Pygame's screen has been created
while True:
    name = event.wait()         # wait for an event
    if name.type == QUIT:
        pygame.quit()           # exit the game
        break
    elif name.type == type:
        code to handle another type of events
    ...

    code to update/redraw the game between events
```

Mouse Clicks

- When the user presses a mouse button, you get events with a type of `MOUSEBUTTONDOWN` and `MOUSEBUTTONUP`.
  - mouse movement is a `MOUSEMOTION` event

- `mouse.get_pos()` returns the mouse cursor's current position as an (x, y) tuple

Example:

```
ev = event.wait()
if ev.type == MOUSEBUTTONDOWN:
    # user pressed a mouse button
    x, y = mouse.get_pos()
```

Key Presses

- When the user presses a keyboard key, you get events with a type of `KEYDOWN` and then `KEYUP`.
  - event contains `.key` field representing what key was pressed
  - Constants for different keys: `K_LEFT`, `K_RIGHT`, `K_UP`, `K_DOWN`, `K_a` - `K_z`, `K_0` - `K_9`, `K_F1` - `K_F12`, `K_SPACE`, `K_ESCAPE`, `K_LSHIFT`, `K_RSHIFT`, `K_LALT`, `K_RALT`, `K_LCTRL`, `K_RCTRL`, ...

Example:

```
ev = event.wait()
if ev.type == KEYDOWN:
    if ev.key == K_ESCAPE:
        pygame.quit()
```

Collision Detection

- **collision detection**: Noticing whether one sprite or object has touched another, and responding accordingly.
  - A major part of game programming

- In Pygame, collision detection is done by examining sprites, rectangles, and points, and asking whether they intersect.

python™

Rect

- a 2D rectangle associated with each sprite (.rect field)
  - Fields: top, left, bottom, right, center, centerx, centery, topleft, topright, bottomleft, bottomright, width, height, size, ...

Method Name	Description
collidepoint(**p**)	returns True if this Rect contains the point
colliderect(**rect**)	returns True if this Rect contains the rect
contains(**rect**)	returns True if this Rect contains the other
move(**x**, **y**)	moves a Rect to a new position
inflate(**dx**, **dy**)	grow/shrink a Rect in size
union(rect)	joins two Rects

Collision Example

- Detecting whether a sprite touches the mouse cursor:

```
ev = event.wait()
if ev.type == MOUSEBUTTONDOWN:
    if sprite.rect.collidepoint(mouse.get_pos()):
        # then the mouse cursor touches the sprite
        ...
```

- **Exercise**: Detect when the user clicks on the Mole. Make the mole run away by fleeing to a new random location from (0, 0) to (600, 400).

Exercise Solution

```
class Mole(Sprite):
    def __init__(self):
        Sprite.__init__(self)
        self.image = image.load("mole.gif")
        self.rect = self.image.get_rect()

    def flee(self):
        self.rect.left = randint(0, 600)   # random location
        self.rect.top = randint(0, 400)

...

while True:
    ev = event.wait()              # wait for an event
    if ev.type == QUIT:
        pygame.quit()
        break
    elif ev.type == MOUSEBUTTONDOWN:
        if my_mole.rect.collidepoint(mouse.get_pos()):
            my_mole.flee()
    ...
```