JTSK-350112

Advanced Programming in Python

Python II

Lecture 3 & 4

Dr. Kinga Lipskoch

Spring 2018

Agenda Week 2

- Object-oriented programming
    - Exceptions
    - Persistent storage of objects
    - Inheritance and polymorphism
    - Playing card games
- Interactive graphics programming
    - Simple graphics class
    - Mouse clicks
    - Textual input

Exceptions try: ... except ...

- Python indicates errors by raising exceptions
- Exceptions are caught by try ... except blocks

```
try:
  <try_suite>
except <exception_group_1> as <var_1>:
  <except_suite_1>
...
except <exception_group_n> as <var_n>:
  <except_suite_n>
else:                # optional
  <else_suit>        # executed if no exception
finally:             # optional
  <finally_suit>     # always executed at end
```

Reading a Text File without Exceptions

```
f = open("myfile.txt", "r")

while True:
  text = f.readline()
  if text == '':
    break
  print(text)

f.close()
```

Reading a Text File with Exceptions

```
filename = "myfile.txt"

try:
    f = open(filename, "r")
    for line in f:
        print(line)
except: # handle any exception (general version)
    sys.exit("Could not openfile {}".format(
   filename))
else:
    f.close()
```

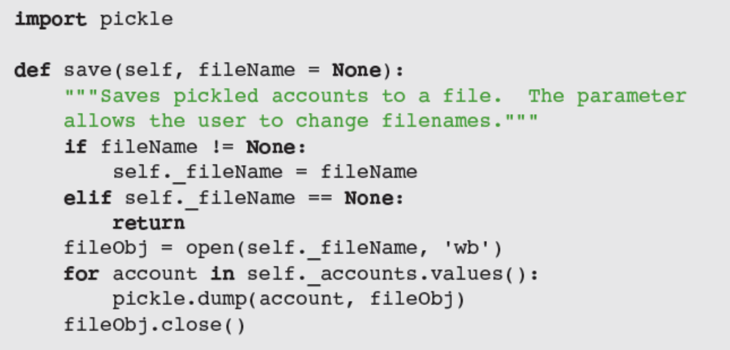
Raising Exceptions and Example

- The `raise` statement allows the programmer to force a specified exception to occur
- The programmer can catch built-in exceptions like `ZeroDivisionError`, `FileNotFoundError`, etc. and can raise and catch own written error objects derived from the parent class called `Exception`
- `exceptions_ex.py`
- `raise_exception.py`

Using `pickle` for Permanent Storage of Objects

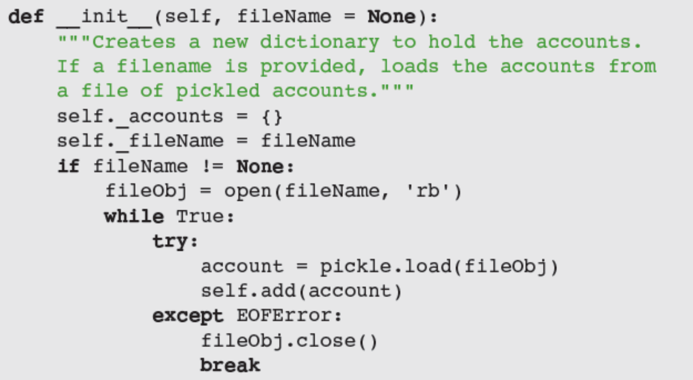
- Name comes from pickling cucumbers
- `pickle` allows the programmer to save and load objects using a process called pickling and unpickling
- Python takes care of all of the conversion details
- Easily allows persistent storage of objects
- `picklestudent.py`

Using pickle.dump() to Save the Accounts in a Bank



```
import pickle

def save(self, fileName = None):
    """Saves pickled accounts to a file.  The parameter
    allows the user to change filenames."""
    if fileName != None:
        self._fileName = fileName
    elif self._fileName == None:
        return
    fileObj = open(self._fileName, 'wb')
    for account in self._accounts.values():
        pickle.dump(account, fileObj)
    fileObj.close()
```

pickle.load() Loads Pickled Objects from a File

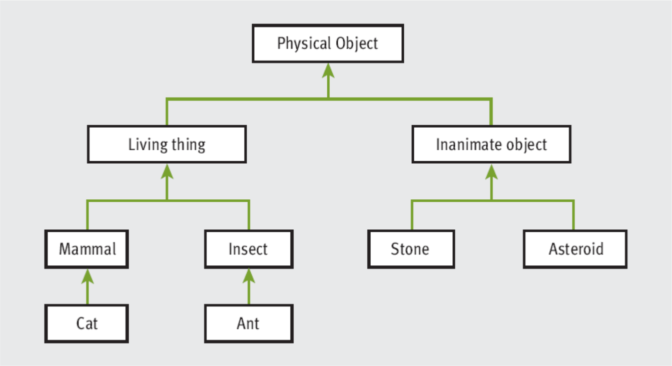


```
def __init__(self, fileName = None):
    """Creates a new dictionary to hold the accounts.
    If a filename is provided, loads the accounts from
    a file of pickled accounts."""
    self._accounts = {}
    self._fileName = fileName
    if fileName != None:
        fileObj = open(fileName, 'rb')
        while True:
            try:
                account = pickle.load(fileObj)
                self.add(account)
            except EOFError:
                fileObj.close()
                break
```

Structuring Classes with Inheritance and Polymorphism

- Most object-oriented languages require the programmer to master the following techniques:
  - Data encapsulation: Restricting manipulation of an object's state by external users to a set of method calls
  - Inheritance: Allowing a class to automatically reuse and extend code of similar but more general classes
  - Polymorphism: Allowing several different classes to use the same general method names
- Python's syntax does not enforce data encapsulation
- Inheritance and polymorphism are built into Python

Inheritance Hierarchies and Modeling (1)



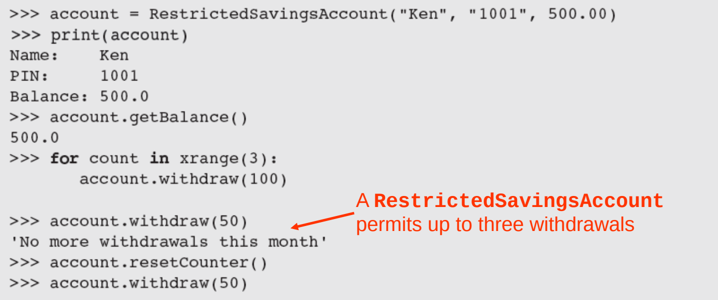
[FIGURE 8.3] A simplified hierarchy of objects in the natural world

Inheritance Hierarchies and Modeling (2)

- In Python, all classes automatically extend the built-in `object` class
- It is possible to extend any existing class:

  ```
  class <new class name>(<existing class name>):
  ```
- Example:
  - `PhysicalObject` would extend `object`
  - `LivingThing` would extend `PhysicalObject`
- Inheritance hierarchies provide an abstraction mechanism that allows the programmer to avoid reinventing the wheel or writing redundant code

Example: A Restricted Savings Account



```
>>> account = RestrictedSavingsAccount("Ken", "1001", 500.00)
>>> print(account)
Name:    Ken
PIN:     1001
Balance: 500.0
>>> account.getBalance()
500.0
>>> for count in xrange(3):
       account.withdraw(100)

>>> account.withdraw(50)
'No more withdrawals this month'
>>> account.resetCounter()
>>> account.withdraw(50)
```

A **`RestrictedSavingsAccount`** permits up to three withdrawals

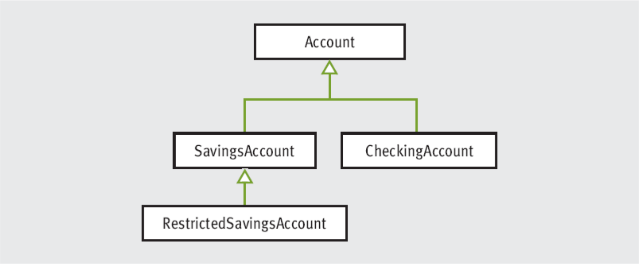
- To call a method in the parent class from within a method with the same name in a subclass:
  `<parent class name>.<method name>(self, <other arguments>)`
- `savings.py`

Python Polymorphism Example

```python
class Bear(object):
    def sound(self):
        print("Groarrr!")
class Dog(object):
    def sound(self):
        print("Woof woof!")
def makeSound(animalType):
    animalType.sound()


bearObj = Bear()
dogObj = Dog()

makeSound(bearObj)
makeSound(dogObj)
```

Abstract Classes

- An abstract class includes data and methods common to its subclasses, but is never instantiated
- Python does not provide a keyword or similar for abstract classes, but it provides a module for Abstract Base Classes (ABC) which is called `abc`



[FIGURE 8.5] An abstract class and three concrete classes

The Costs and Benefits of OOP (1)

- Imperative programming
  - Code consists of I/O, assignment, and control (selection/iteration) statements
  - Does not scale well
- Improvement: Embedding sequences of imperative code in function definitions or subprograms
  - Procedural programming
- Functional programming views a program as a set of cooperating functions
  - No assignment statements

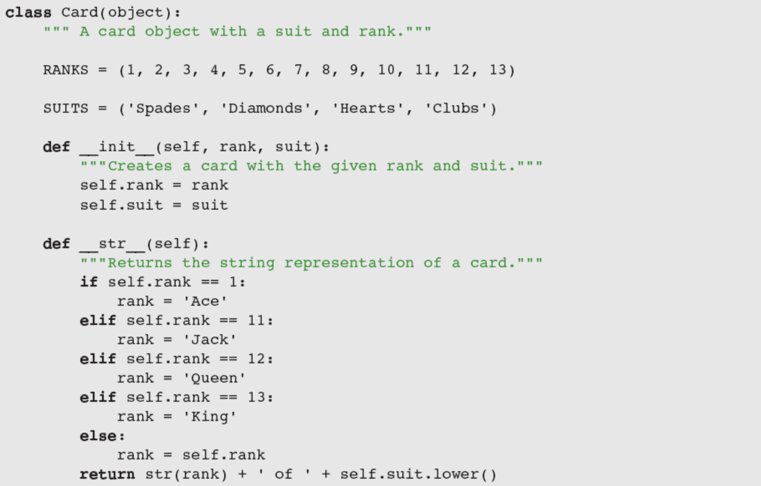
The Costs and Benefits of OOP (2)

- Functional programming does not conveniently model situations where data must change state
- Object-oriented programming attempts to control the complexity of a program while still modeling data that change their state
  - Divides up data into units called objects
  - Well-designed objects decrease likelihood that the system will break when changes are made within a component
  - Can be overused and abused

Playing Cards (1)

- Standard deck has 52 cards
- Four suits:
  - spades ♠, hearts ♡, diamonds ♢, clubs ♣
  - each suit has 13 cards
  - ace, 2 – 10, jack, queen, king
- `Card` class
- Card has two attributes: `rank` and `suit`
- Class attributes: set of all suits, set of all ranks

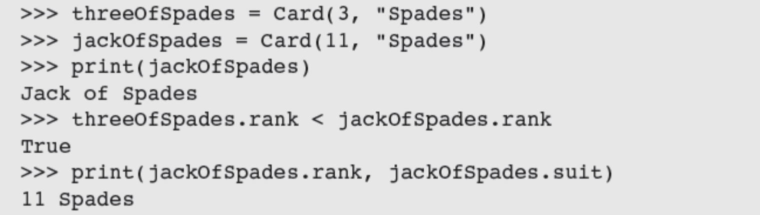
Playing Cards (2)



```
class Card(object):
    """ A card object with a suit and rank."""

    RANKS = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)

    SUITS = ('Spades', 'Diamonds', 'Hearts', 'Clubs')

    def __init__(self, rank, suit):
        """Creates a card with the given rank and suit."""
        self.rank = rank
        self.suit = suit

    def __str__(self):
        """Returns the string representation of a card."""
        if self.rank == 1:
            rank = 'Ace'
        elif self.rank == 11:
            rank = 'Jack'
        elif self.rank == 12:
            rank = 'Queen'
        elif self.rank == 13:
            rank = 'King'
        else:
            rank = self.rank
        return str(rank) + ' of ' + self.suit.lower()
```

Playing Cards (3)

- Use of the Card class:

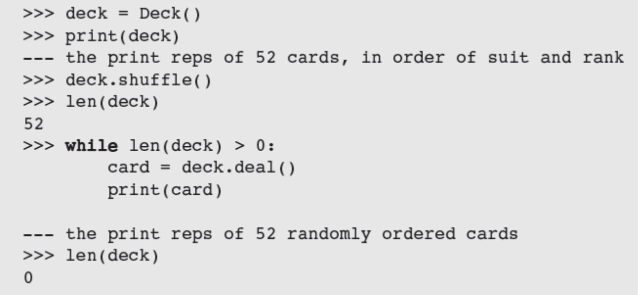


```
>>> threeOfSpades = Card(3, "Spades")
>>> jackOfSpades = Card(11, "Spades")
>>> print(jackOfSpades)
Jack of Spades
>>> threeOfSpades.rank < jackOfSpades.rank
True
>>> print(jackOfSpades.rank, jackOfSpades.suit)
11 Spades
```

- Because the attributes are only accessed and never modified, we do not include any methods other than `__str__` for string representation
- A card is little more than a container of two data values

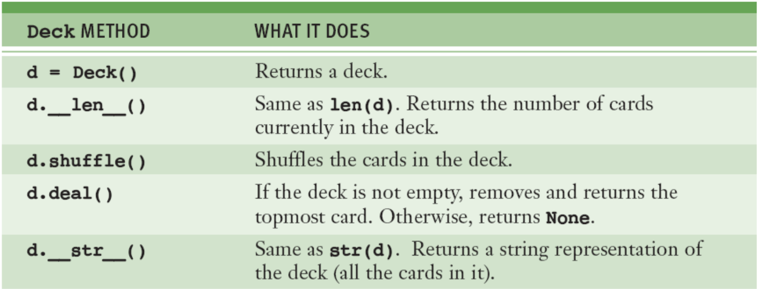
Playing Cards (4)

- Unlike an individual card, a deck has significant behavior that can be specified in an interface
- One can shuffle the deck, deal a card, and determine the number of cards left in it



```
>>> deck = Deck()
>>> print(deck)
--- the print reps of 52 cards, in order of suit and rank
>>> deck.shuffle()
>>> len(deck)
52
>>> while len(deck) > 0:
        card = deck.deal()
        print(card)

--- the print reps of 52 randomly ordered cards
>>> len(deck)
0
```

Playing Cards (5)



Deck METHOD	WHAT IT DOES
`d = Deck()`	Returns a deck.
`d.__len__()`	Same as **`len(d)`**. Returns the number of cards currently in the deck.
`d.shuffle()`	Shuffles the cards in the deck.
`d.deal()`	If the deck is not empty, removes and returns the topmost card. Otherwise, returns **`None`**.
`d.__str__()`	Same as **`str(d)`**. Returns a string representation of the deck (all the cards in it).

[TABLE 8.7] The interface for the **Deck** class

- During instantiation, all 52 unique cards are created and inserted into the deck's internal list
- `cards.py`

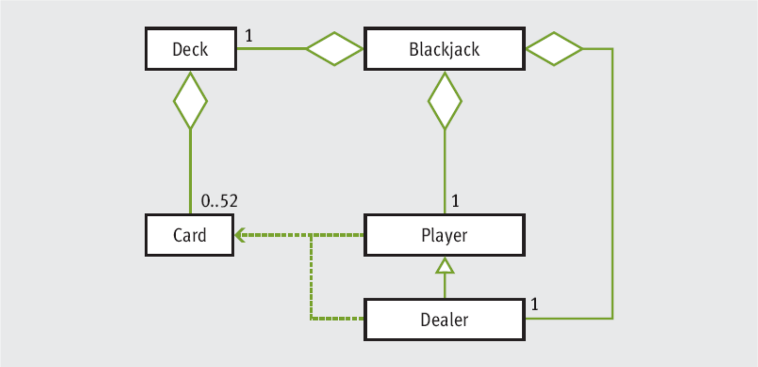
Blackjack (1)

- Player and dealer try to get 21 points
- Numbered cards: face value
- Ace counts 1 or 11 points
- Jack, queen, king count 10 points
- Dealer plays against player
- Each player (incl. dealer) receives two cards
- One of the dealer's cards is hidden
- Both may take additional cards

Blackjack (2)

- If player exceeds 21 points, player is busted
- After the player has taken cards, dealer reveals the hidden card and will take additional cards as long as the total is 16 or less
- If dealer busts, the player still in the game (not busted) wins
- Else
  - player $>$ dealer: player wins
  - player $<$ dealer: player loses
  - player $==$ dealer: tie (a.k.a. pushing)

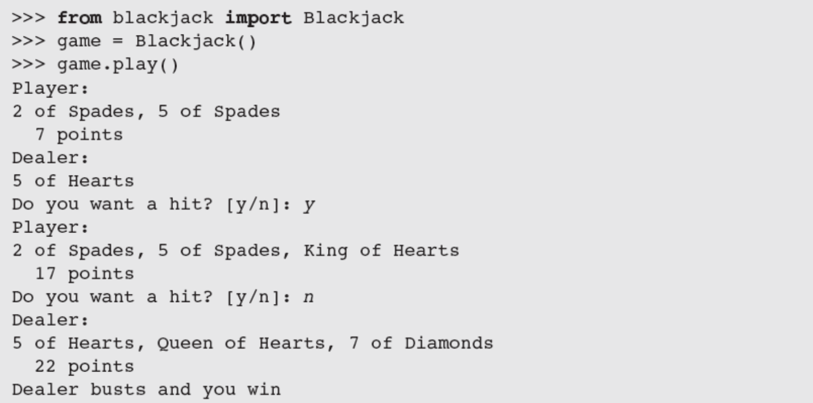
Example: The Dealer and a Player in the Game of Blackjack (1)



[FIGURE 8.4] The classes in the blackjack game application

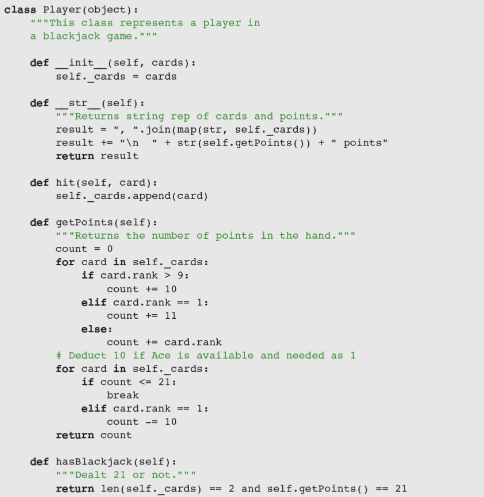
Example: Blackjack (2)

An object belonging to the `Blackjack` class sets up the game and manages the interactions with user



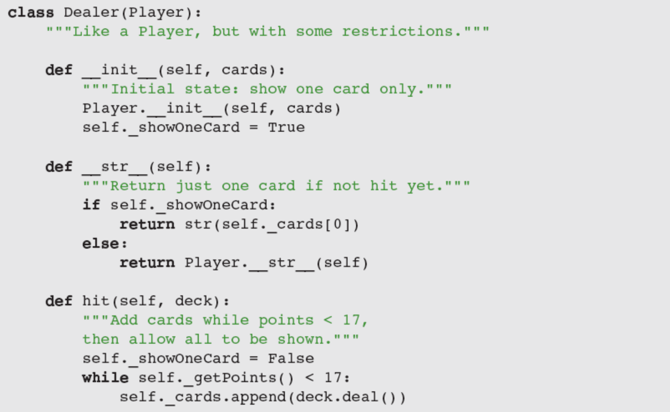
```
>>> from blackjack import Blackjack
>>> game = Blackjack()
>>> game.play()
Player:
2 of Spades, 5 of Spades
  7 points
Dealer:
5 of Hearts
Do you want a hit? [y/n]: y
Player:
2 of Spades, 5 of Spades, King of Hearts
  17 points
Do you want a hit? [y/n]: n
Dealer:
5 of Hearts, Queen of Hearts, 7 of Diamonds
  22 points
Dealer busts and you win
```

Example: Blackjack (3)



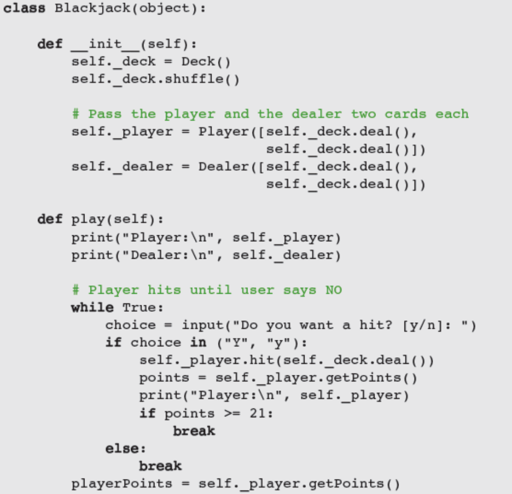
```
class Player(object):
    """This class represents a player in
    a blackjack game."""

    def __init__(self, cards):
        self._cards = cards

    def __str__(self):
        """Returns string rep of cards and points."""
        result = ", ".join(map(str, self._cards))
        result += "\n  " + str(self.getPoints()) + " points"
        return result

    def hit(self, card):
        self._cards.append(card)

    def getPoints(self):
        """Returns the number of points in the hand."""
        count = 0
        for card in self._cards:
            if card.rank > 9:
                count += 10
            elif card.rank == 1:
                count += 11
            else:
                count += card.rank
        # Deduct 10 if Ace is available and needed as 1
        for card in self._cards:
            if count <= 21:
                break
            elif card.rank == 1:
                count -= 10
        return count

    def hasBlackjack(self):
        """Dealt 21 or not."""
        return len(self._cards) == 2 and self.getPoints() == 21
```

Example: Blackjack (4)



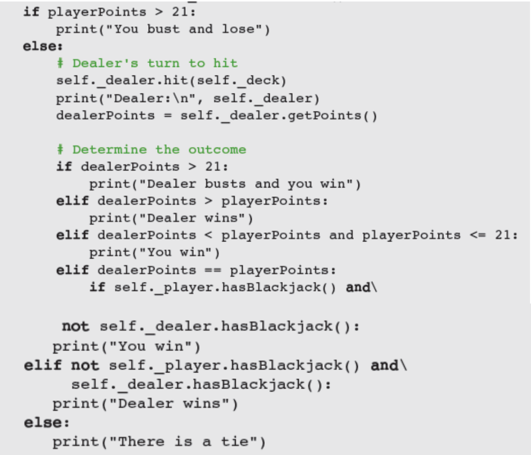
```python
class Dealer(Player):
    """Like a Player, but with some restrictions."""

    def __init__(self, cards):
        """Initial state: show one card only."""
        Player.__init__(self, cards)
        self._showOneCard = True

    def __str__(self):
        """Return just one card if not hit yet."""
        if self._showOneCard:
            return str(self._cards[0])
        else:
            return Player.__str__(self)

    def hit(self, deck):
        """Add cards while points < 17,
        then allow all to be shown."""
        self._showOneCard = False
        while self._getPoints() < 17:
            self._cards.append(deck.deal())
```

Example: Blackjack (5)



```
class Blackjack(object):

    def __init__(self):
        self._deck = Deck()
        self._deck.shuffle()

        # Pass the player and the dealer two cards each
        self._player = Player([self._deck.deal(),
                               self._deck.deal()])
        self._dealer = Dealer([self._deck.deal(),
                               self._deck.deal()])

    def play(self):
        print("Player:\n", self._player)
        print("Dealer:\n", self._dealer)

        # Player hits until user says NO
        while True:
            choice = input("Do you want a hit? [y/n]: ")
            if choice in ("Y", "y"):
                self._player.hit(self._deck.deal())
                points = self._player.getPoints()
                print("Player:\n", self._player)
                if points >= 21:
                    break
            else:
                break
        playerPoints = self._player.getPoints()
```

Example: Blackjack (6)



```
if playerPoints > 21:
    print("You bust and lose")
else:
    # Dealer's turn to hit
    self._dealer.hit(self._deck)
    print("Dealer:\n", self._dealer)
    dealerPoints = self._dealer.getPoints()

    # Determine the outcome
    if dealerPoints > 21:
        print("Dealer busts and you win")
    elif dealerPoints > playerPoints:
        print("Dealer wins")
    elif dealerPoints < playerPoints and playerPoints <= 21:
        print("You win")
    elif dealerPoints == playerPoints:
        if self._player.hasBlackjack() and\

    not self._dealer.hasBlackjack():
   print("You win")
elif not self._player.hasBlackjack() and\
     self._dealer.hasBlackjack():
   print("Dealer wins")
else:
   print("There is a tie")
```

blackjack.py

Object-oriented Design (OOD)

Object-oriented design is the process of trying to develop a set of classes to solve a problem

- Look for object candidates
- Identify instance and class variables
- Consider interfaces, what does the object need to support?
- Design step-by-step
- Try out alternatives
- Try to keep things simple

Summary (1)

- A simple class definition consists of a header and a set of method definitions
- In addition to methods, a class can also include instance variables
- Constructor or `__init__` method is called when a class is instantiated
- A method contains a header and a body
- An instance variable is introduced and referenced like any other variable, but is always prefixed with `self`

Summary (2)

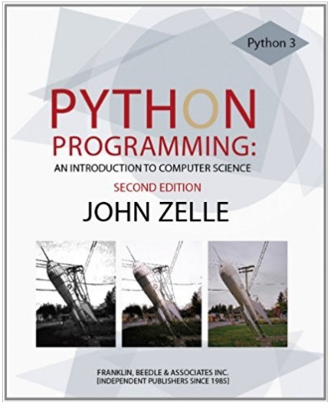
- Some standard operators can be overloaded for use with new classes of objects
- When a program can no longer reference an object, it is considered dead and its storage is recycled by the garbage collector
- A class variable is a name for a value that all instances of a class share in common
- Pickling is the process of converting an object to a form that can be saved to permanent file storage
- `try ... except` statement is used to catch and handle exceptions

Summary (3)

- Most important features of OO programming: encapsulation, inheritance, and polymorphism
  - Encapsulation restricts access to an object's data to users of the methods of its class – not forced in Python
  - Inheritance allows one class to pick up the attributes and behavior of another class "for free"
  - Polymorphism allows methods in several different classes to have the same headers
- A data model is a set of classes that are responsible for managing the data of a program

Simple Graphics

- Slides sources
- John Zelle: Python Programming, Franklin, Beedle & Associates, 2010, Second Edition, ISBN 978-1-59028-241-0
- Simple graphics library (`graphics.py`), which is based on `tkinter`


Python 3
PYTHON PROGRAMMING:
AN INTRODUCTION TO COMPUTER SCIENCE
SECOND EDITION
JOHN ZELLE
FRANKLIN, BEEDLE & ASSOCIATES INC.
[INDEPENDENT PUBLISHERS SINCE 1985]

Simple Graphics Programming

- Uses the `graphics.py` library supplied with the additional materials
- Two location choices to place this file:
  - In Python's lib directory with other libraries
  - In the folder of your graphics related programs
- Since this is a library, we need to import the graphics commands

  ```
  import graphics
  ```

- A graphics window is a place and holder on the screen where the graphics will appear

  ```
  win = graphics.GraphWin()
  ```

- New window titled "`Graphics Window`" is created

Window Objects (1)

- `GraphWin` is an object assigned to the variable `win`
- We can manipulate the window object through this variable, similar to manipulating files through file variables
- Windows can be closed/destroyed by issuing the command `win.close()`
- No need to use `graphics.` when doing import like this

```python
from graphics import *
win = GraphWin()
```

Window Objects (2)

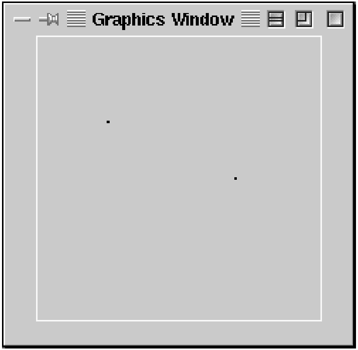
- A graphics window is a collection of points called pixels (picture elements)
- The default `GraphWin` is 200 pixels tall by 200 pixels wide ($40,000$ pixels total)
- One way to get pictures into the window is one pixel at a time, which would be tedious
- The graphics routine has a number of predefined routines to draw geometric shapes

Point Objects (1)

- The simplest object is the `Point`
- Like points in geometry, point locations are represented with a coordinate system (`x`, `y`), where `x` is the horizontal location of the point and `y` is the vertical location
- The origin (0, 0) in a graphics window is the upper left corner
- `x` values increase from right to left, `y` values from top to bottom
- The lower right corner is (199, 199)

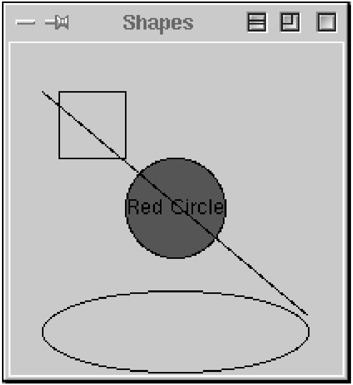
Point Objects (2)

```
>>> p = Point(50, 60)
>>> p.getX()
50
>>> p.getY()
60
>>> win = GraphWin()
>>> p.draw(win)
>>> p2 = Point(140, 100)
>>> p2.draw(win)
```



More Shapes/Objects

```
>>> ### Open a graphics window
>>> win = GraphWin('Shapes')
>>> ### Draw a red circle centered at point
    (100, 100) with radius 30
>>> center = Point(100, 100)
>>> circ = Circle(center, 30)
>>> circ.setFill('red')
>>> circ.draw(win)
>>> ### Put a textual label in the center of
    the circle
>>> label = Text(center, "Red Circle")
>>> label.draw(win)
>>> ### Draw a square using a Rectangle object
>>> rect = Rectangle(Point(30, 30), Point(70,
    70))
>>> rect.draw(win)
>>> ### Draw a line segment using a Line object
>>> line = Line(Point(20, 30), Point(180, 165))
>>> line.draw(win)
>>> ### Draw an oval using the Oval object
>>> oval = Oval(Point(20, 150), Point(180,
    199))
>>> oval.draw(win)
```



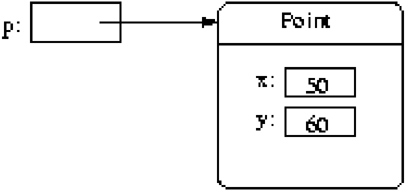
Data Storage in Objects (1)

- Computation is preformed by asking an object to carry out one of its operations
- In the previous example we manipulated `GraphWin`, `Point`, `Circle`, `Oval`, `Line`, `Text` and `Rectangle`
- These are examples of classes in `graphics.py`
- The constructor for the `Point` class requires to parameters, the `x` and `y` coordinates for the point

  ```
  p = Point(50, 60)
  ```
- These values are stored as instance variables inside of the object

Data Storage in Objects (2)

Only the most relevant instance variables are shown (others include the color, window they belong to, etc.)



Accessing Data of Objects

- To perform an operation on an object, we send the object a message
- The set of messages an object responds to are called the methods of the object
- By sending the object the message we are actually invoking the method
- `p.getX()` and `p.getY()` return the `x` and `y` coordinates of the point
- Routines like these are referred to as accessors because they allow us to access information from the instance variables of the object

Changing Data of Objects

- Other methods change the state of the object by changing the values of the object's instance variables
- Methods that change the state of an object are called mutators
- `move(dx, dy)` moves the object `dx` units in the `x` direction and `dy` in the `y` direction
- `move` erases the old image and draws it in its new position

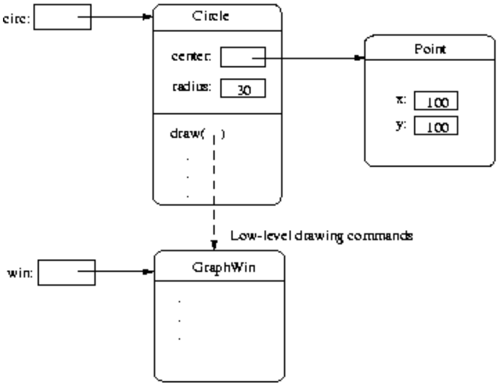
Drawing Shape Objects in the Window (1)

```
circ = Circle(Point(100, 100), 30)
win = GraphWin()
circ.draw(win)
```

- The first line creates a circle with radius 30 centered at position (100, 100)
- We used the `Point` constructor to create a location for the center of the circle
- The last line is a request to the `Circle` object `circ` to draw itself into the `GraphWin` object `win`

Drawing Shape Objects in the Window (2)

The `draw` method uses information about the center and radius of the circle from the instance variable



Object Aliasing (1)

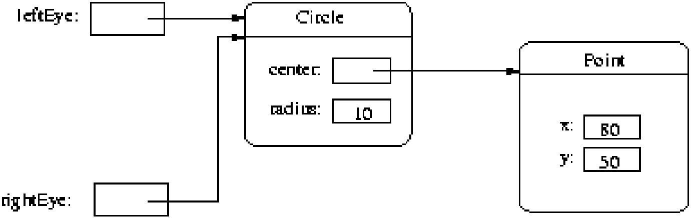
- It is possible for two different variables to refer to the same object - changes made to the object through one variable will be visible to the other

```
leftEye = Circle(Point(80,50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
rightEye = leftEye
rightEye.move(20,0)
```

- The idea is to create the left eye and copy that to the right eye which gets moved 20 units

Object Aliasing (2)

- The assignment `rightEye = leftEye` makes `rightEye` and `leftEye` refer to the same circle
- The situation where two variables refer to the same object is called aliasing



Object Aliasing (3)

- There are two ways to get around this
- We could make two separate circles, one for each eye:

```
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
rightEye = Circle(Point(100, 50), 5)
rightEye.setFill('yellow')
rightEye.setOutline('red')
```

Object Cloning

- The graphics library has a better solution
- Graphical objects have a clone method that will make a copy of the object

```python
# Correct way to create two circles, using clone
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
rightEye = leftEye.clone()
# rightEye is an exact copy of the left
rightEye.move(20, 0)
```

Interactive Graphics (1)

- In a GUI environment, users typically interact with their applications by clicking on buttons, choosing items from menus, and typing information into on-screen text boxes
- Event-driven programming draws interface elements (widgets) on the screen and then waits for the user to do something

Interactive Graphics (2)

- An event is generated whenever a user moves the mouse, clicks the mouse, or types a key on the keyboard
- An event is an object that encapsulates information about what just happened
- The event object is sent to the appropriate part of the program to be processed, for example, a button event
- The graphics module hides the underlying, low-level window management and provides two simple ways to get user input in a `GraphWin`

Getting Mouse Clicks (1)

- We can get graphical information from the user via the `getMouse()` method of the `GraphWin` class
- When `getMouse()` is invoked on a `GraphWin`, the program pauses and waits for the user to click the mouse somewhere in the window
- The spot where the user clicked is returned as a `Point`

Getting Mouse Clicks (2)

- The following code reports the coordinates of a mouse click
- We can use the accessors like getX() and getY() or other methods on the point returned

```
from graphics import *
win = GraphWin("Click Me!")
p = win.getMouse()
print("You clicked", p.getX(), p.getY())
```

Moving a Circle

movecircle.py

```python
from graphics import *
def main():
    win = GraphWin()
    shape = Circle(Point(50, 50), 20)
    shape.setOutline("red")
    shape.setFill("red")
    shape.draw(win)
    for i in range(10):
        p = win.getMouse()
        c = shape.getCenter()
        dx = p.getX() - c.getX()
        dy = p.getY() - c.getY()
        shape.move(dx, dy)
    win.close()
main()
```

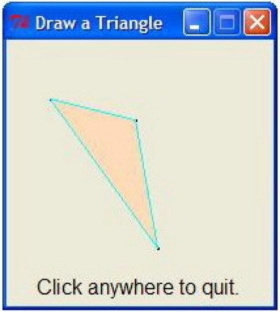
Getting Mouse Clicks (3)

```
# triangle.pyw - Interactive graphics program

from graphics import *
def main():
    win = GraphWin("Draw a Triangle")
    win.setCoords(0.0, 0.0, 10.0, 10.0)
    message = Text(Point(5, 0.5), "Click on three points")
    message.draw(win)

    # Get and draw three vertices of triangle
    p1 = win.getMouse()
    p1.draw(win)
    p2 = win.getMouse()
    p2.draw(win)
```

Getting Mouse Clicks (4)

```
    p3 = win.getMouse()
    p3.draw(win)

    # Use Polygon object to draw the triangle
    triangle = Polygon(p1,p2,p3)
    triangle.setFill("peachpuff")
    triangle.setOutline("cyan")
    triangle.draw(win)

    # Wait for another click to exit
    message.setText("Click anywhere to quit.")
    win.getMouse()
    win.close()

main()
```

Result of Interactive Drawing



Click anywhere to quit.

`triangle.py`

The pyw Extension

- If you are programming in a Windows environment, using the `.pyw` extension for your file will cause the Python shell window to not display when you double-click the program icon

Drawing a Triangle

- There is no triangle class
- Use the general polygon class, which takes any number of points and connects them into a closed shape
- If you have three points, creating a triangle polygon: `triangle = Polygon(p1, p2, p3)`
- A single text object is created and drawn near the beginning of the program

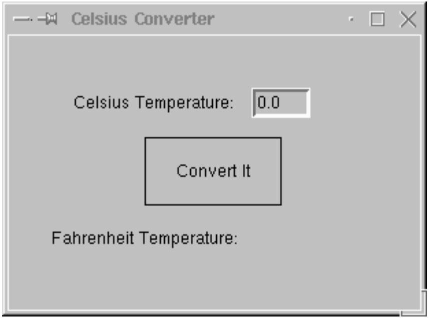
```
message = Text(Point(5,0.5), "Click on three
    points")
message.draw(win)
```

- To change the prompt, just change the text to be displayed `message.setText("Click anywhere to quit.")`

Handling Textual Input

- The triangle program's input was done completely through mouse clicks
- There's also an `Entry` object that can get keyboard input
- The `Entry` object draws a box on the screen that can contain text
- It understands `setText()` and `getText()`, with one difference that the input can be edited

Example of Textual Input



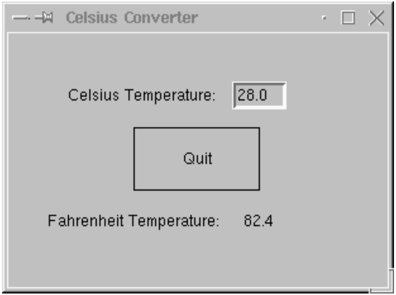
Programming Textual Input within a Window (1)

```python
# convert_gui.pyw
# Program to convert Celsius to Fahrenheit using a simple
#   graphical interface.

from graphics import *

def main():
    win = GraphWin("Celsius Converter", 300, 200)
    win.setCoords(0.0, 0.0, 3.0, 4.0)
    # Draw the interface
    Text(Point(1,3), "   Celsius Temperature:").draw(win)
    Text(Point(1,1), "Fahrenheit Temperature:").draw(win)
    input = Entry(Point(2, 3), 5)
    input.setText("0.0")
    input.draw(win)
    output = Text(Point(2.3, 1),"")
    output.draw(win)
```

Programming Textual Input within a Window (2)

```
    button = Text(Point(1.5, 2.0),"Convert It")
    button.draw(win)
    Rectangle(Point(1, 1.5), Point(2, 2.5)).draw(win)

    # wait for a mouse click
    win.getMouse()
    # convert input
    celsius = eval(input.getText())
    fahrenheit = 9.0/5.0 * celsius + 32
    # display output and change button
    output.setText(fahrenheit)
    button.setText("Quit")
    # wait for click and then quit
    win.getMouse()
    win.close()

main()
```

Result of Running



Celsius Converter

Celsius Temperature: 28.0

Fahrenheit Temperature: 82.4

`convert_gui.py`

Comments (1)

- When run, this program produces a window with an entry box for typing in the Celsius temperature and a button to "do" the conversion
- The button is for show only
- We are just waiting for a mouse click anywhere in the window

Comments (2)

- Initially, the input entry box is set to contain `0.0`
- The user can delete this value and type in another value
- The program pauses until the user clicks the mouse - we do not care where so we do not store the point
- The input is processed in three steps:
  - The value entered is converted into a number with `eval`
  - This number is converted to degrees Fahrenheit
  - This number is then converted to a string and formatted for display in the output text area