

# Programming Concepts II: Object-Oriented Programming

QF 205 L9



# Learning Objectives

- This lecture is the third of three lectures that talks about the fundamentals of Python programming
- In this lecture, we learn about the very important programming methodology called Object Oriented Programming (OOP)
- OOP promotes code reuse and more direct approaches to the modelling of problem situations
- We also include a section on exception handling

# Contents

- Classes Concepts
- Classes Example
  - Market Indices
  - Tkinter
    - Appendix: Tkinter Widgets
- Technical Aside: Python Class Mechanism
- Exceptions
- Exercises
- References

# Classes Concepts

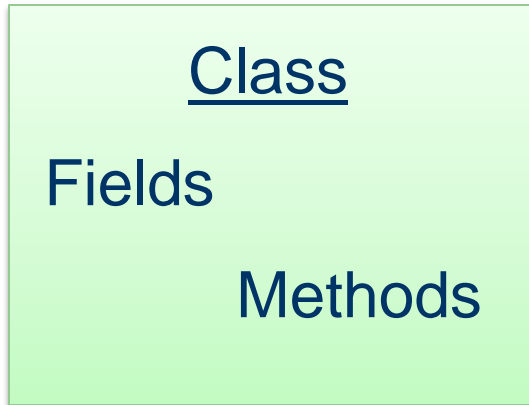
# Classes and OOP

- Classes are part of the Object Oriented Programming paradigm
- Four basic principles of OOP are:
  - Encapsulation: packaging of fields and methods into classes
  - Inheritance: corresponds to specialization of concepts (animals->animals that fly)
  - Reusability: inheritance, instantiation achieve this
  - Polymorphism: one name/operator, different meanings
- OOP achieves two goals
  - It corresponds more closely to real problems (things are classes and objects)
  - It allows code to be reused

# Modelling with Classes

- How to think?
  - Classes are templates (e.g. animals)
  - Instances (objects) are specializations (e.g. elephants, sharks, eagles)
  - Inheritance are categorical specializations (e.g. animals that fly)
- Computer programs are good for performing tasks which are repetitive, which involves huge quantities of data, which have clear and unambiguous steps...
- OOP is a paradigm of programming....how to use this to solve problems?
  - When a concept keep recurring, and the concept is a composite creature that has parts in the form of attributes, and there are some similar ways to work with this concept
  - Examples: Equity Index, GUI with Tkinter

# Modelling with Classes

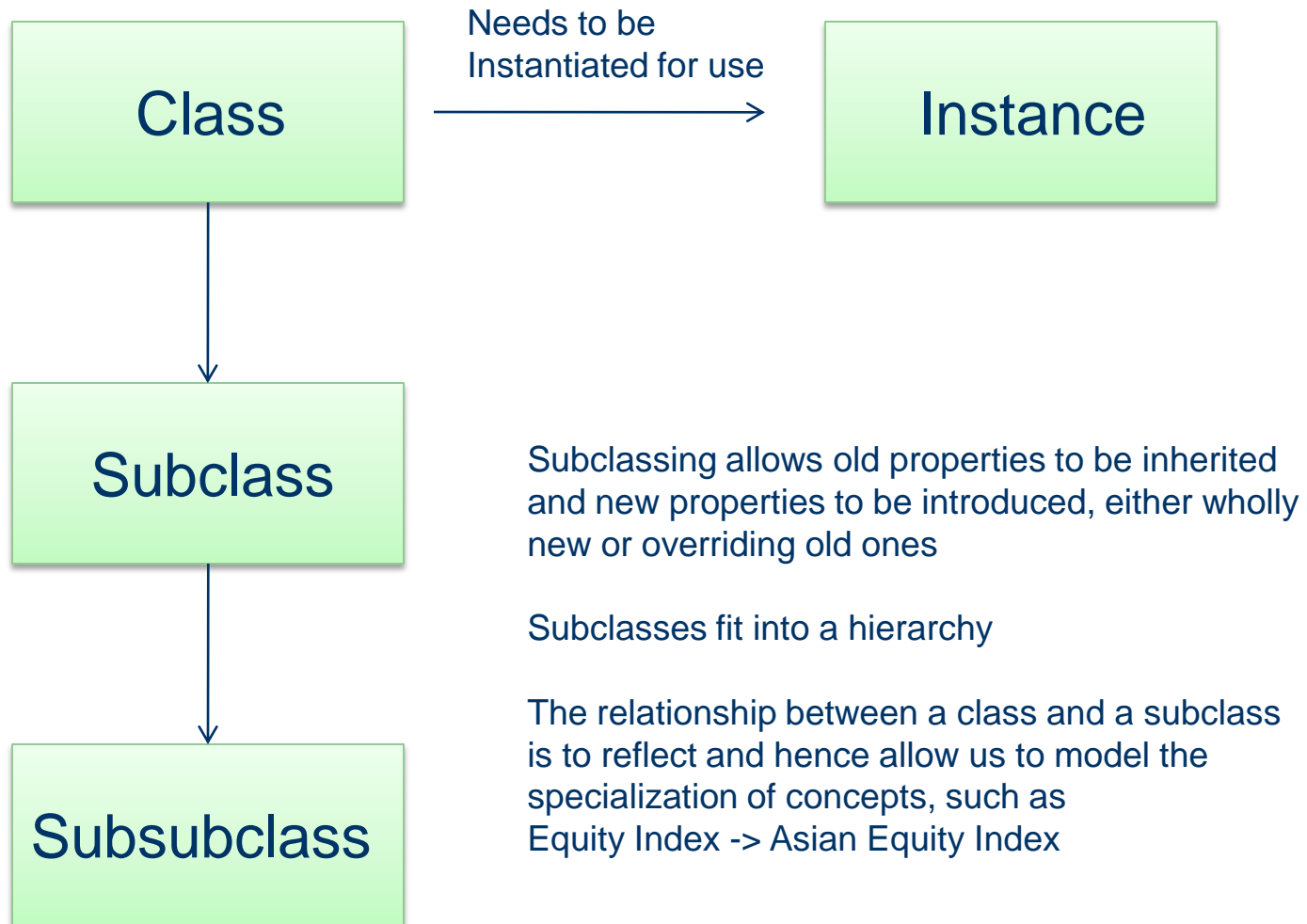


A class is a “package” of fields and methods (collectively: attributes)

Examples:

- 1) Equity Index
  - a) Attributes
    - Component stocks
    - Market
  - b) Methods
    - Index Level on Date
  
- 2) GUI
  - a) Attributes
    - Width and Height
    - Background color
    - Button
  - b) Methods
    - Exit
    - Add child

# Dynamics of Classes



# Topics

- Definition of a class
- Constructor: `__init__`
- `self`
  - Refers to the instance being processed
- Creating and using an instance of a class (an object)
- Operator overloading
- Inheritance
- Mix-ins

# Adding fields to class and instance

```
class SimplestClass:
    pass

# Creating an instance
X = SimplestClass() # nothing to do with it
Z = SimplestClass()

# Fields can be attached
SimplestClass.name = 'Bob'
SimplestClass.age = 40

print SimplestClass.name
print X.name # This prints Bob too - why does this happen?

Y = SimplestClass()
Y.name = 'Mary'
print Y.name # Mary
print X.name # Bob
print SimplestClass.name # Bob
```

```
X.name = 'Tom'
print Y.name # Mary
print X.name # Tom
print SimplestClass.name # Bob
```

```
SimplestClass.name = 'Peter'
print Y.name # Mary
print X.name # Tom
print SimplestClass.name # Peter
print Z.name # Peter
```

```
print SimplestClass.__dict__.keys()
print X.__dict__.keys()
print Y.__dict__.keys()
print Z.__dict__.keys()
```

# What's going on?

- Classes are prototypes – they spawn instances
- Once an instance of a class is created, a memory location is reserved for this object...but its type is still linked to the class from which it is derived
- When fields of an instance are modified, nothing happens to the class
- When fields of a class are modified, one of two things happens:
  - If the field already exists, the instances are untouched
  - If the field does not already exist, this field is transmitted to its instances. An instance will refer to this field as long as its copy of this field has not been modified.
- Use the `keys()` method of the `__dict__` field to check which fields have been defined
  - `SimplestClass.__dict__.keys()`
  - `X.__dict__.keys()`

# Class definition and attributes (fields and methods) –

```
class ExampleClass:
    data = 'spam'          # field
    numInstances = 0
    def __init__(self,value): # constructor
        self.data = value   # instance attribute
        ExampleClass.numInstances += 1 # class attribute
    def display(self):      # method
        print self.data, ExampleClass.data # instance and class attrs

def printNumInstances():
    print "Number of instances created: ", ExampleClass.numInstances

a = ExampleClass(5)
b = ExampleClass(6)
c = ExampleClass(7)
a.display()
b.display()
c.display()
printNumInstances()
```

# Inheritance

```
class Super:
    def method(self):
        print 'in Super.method'

# inheritance
class Sub(Super):
    def method(self):          # overriding a method
        print 'starting Sub.method' # new action
        Super.method(self)      # run default action (from super class)
        print 'ending Sub.method' # new action

x = Super()
x.method()
y = Sub()
y.method()
```

# Operator Overloading

```
class Number:  
    def __init__(self,start):  
        self.data = start  
    def __sub__(self,other):  
        return Number(self.data-other)
```

```
x = Number(5)  
y = x - 2    # the operator - is overloaded by __sub__ - interpret as __sub__(x,2)  
print y.data
```

# Some common operator overloading methods

- `__init__`
  - Overloads: constructor
  - Calls: `X = Class()`
- `__add__`
  - Overloads: operator `+`
  - Calls: `X+Y`
- `__call__`
  - Overloads: function class
  - Calls: `X()`
- `__len__`
  - Overloads: length
  - Calls: `len(X)`

For further info:

<http://docs.python.org/reference/datamodel.html>

# operatoroverloading

```
class Number:
    def __init__(self,start):
        self.data = start
    def __add__(self,other):
        return Number(self.data-other)
    def __call__(self,val):
        print 'The value is: ', val
    def __len__(self):
        return 42
```

```
x = Number(5) # __init__
y = x + 2     # __add__
print y.data
x(9)         # __call__
print len(x) # __len__
```

# Examples

# A class is defined with 2 attribute methods

```
class FirstClass:
    def setdata(self,value):
        self.data = value
    def display(self):
        print self.data
```

```
x = FirstClass()
x.setdata('King Arthur')
x.display()
y = FirstClass()
y.setdata('1234')
y.display()
```

# Inheritance

```
class SecondClass(FirstClass):
    def display(self):
        print 'Current value = "%s"' % self.data
```

```
z = SecondClass()
z.setdata(42)
z.display()
```

# \_\_init\_\_ method is the constructor

# \_\_add\_\_ overloads +

# \_\_mul\_\_ overloads \*

```
class ThirdClass(SecondClass):
    def __init__(self,value):
        self.data = value
    def __add__(self,other):
        return ThirdClass(self.data + other)
    def __mul__(self,other):
        self.data = self.data * other
```

```
a = ThirdClass('abc') # the constructor is invoked
```

```
a.display()
```

```
b = a + 'xyz' # __add__ is invoked
```

```
b.display()
```

```
a * 3
```

```
a.display() # __mul__ is invoked
```

# Multiple inheritance and mix-in classes

- In a class statement, more than one superclass can be listed in the parenthesis – the resulting class can inherit from more than one super class
- A class that inherits from more than one class is called a mix-in
- A mix-in inherits various methods from various classes
- Issue: What if there are name collisions?

# mixins

```
class Class1:
    def method1(self):
        print 'method1'

class Class2:
    def method2(self):
        print 'method2'

class Class3(Class1,Class2):
    def method3(self):
        print 'method3 added method'
        Class1.method1(self)
        Class2.method2(self)

X = Class3()
X.method3()
```

# Name Collisions

- If a name is being shared in the parent classes of a mixin, its value is defined by the latest invocation

```
# Exploring field name collision
```

```
class A():  
    def f(self):  
        self.x = 1
```

```
class B():  
    def g(self):  
        self.x = 2
```

```
class C(A,B):  
    pass
```

```
a = C()  
a.f()  
print a.x # 1  
a.g()  
print a.x # 2  
a.f()  
print a.x # 1
```

```
# Exploring method name collision
```

```
class A():  
    def f(self):  
        self.x = 1
```

```
class B():  
    def f(self):  
        self.x = 2
```

```
class C(A,B):  
    pass
```

```
class D(B,A):  
    pass
```

```
c = C()  
c.f()  
print c.x # 1  
d = D()  
d.f()  
print d.x # 2
```

# Python New-Style Classes

- Starting from Python 2.2, classes are of 2 types – the classical ones and what are called “new-style”
- New-style classes are derived from the type object:

```
class MyClass(object):  
    def __init__(self,arg):  
    .....
```

- New-style classes are improvements over the classical ones but for most beginner purposes, they are identical. Starting from Python 3.0, classical classes will completely give way to new-style classes and the peculiar way of declaring them will not be necessary

# Classes Example: Market Indices

# Equity Index

- There are many equity indices
- Each comprises its component stocks
- Each has an index level (on a certain date)
- Each is associated with a certain geographical region
- We'll model this with a class
  - Component stocks and geographical region – defined in the constructor `__init__`
  - Index level – a class method
  - Geographical region - field

# EquityIndex Class

```
def getYF(symbol,dt):
    """
    To-be-implemented function to
    grab price on date dt from YF
    """
    price = 1.0
    return price

class EquityIndex(object):
    """
    This is a simple example
    that illustrates how classes
    may be used to solve
    a practical problem
    """

    def __init__(self,symbollist,geog):
        self.symbols = symbollist
        self.geography = geog

    def index_on(self,dt):
        pricelist = []
        for symbol in self.symbols:
            pricelist.append(getYF(symbol,dt))

        return float(sum(pricelist)) / len(pricelist)
```

Usage:

```
import equityindex as ei
X = ei.EquityIndex(['c','goog'],'US')
print X.geography
print X.index_on('17042010')
```

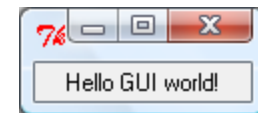
# Classes Example: Tkinter

# What's Tkinter

- Tkinter is a standard way to implement portable user interfaces in Python
- It comprises a collection of widgets and a method of organizing them into user interfaces
- It has the following extension widget sets: Pmw, Tix
- The Python interpreter IDLE (which comes together with the Python installation) is written with Tkinter

# Hello World in 4 lines

```
from Tkinter import Label
widget = Label(None, text='Hello GUI world!')
widget.pack()
widget.mainloop()
```



This already illustrates the basics of a Tkinter application:

- a) Load widget class from Tkinter module
- b) Make an instance of the imported widget class
- c) Pack the widget class within its parent widget (layout management)
- d) Call mainloop to bring up the window and start the Tkinter event loop

# Alternatives

```
import Tkinter
widget = Tkinter.Label(None, text='Hello GUI world!')
widget.pack()
widget.mainloop()
```

Need to prefix Tkinter.  
to methods - tedious

```
from Tkinter import *
root = Tk()
Label(root, text='Hello GUI world!').pack(side=TOP)
root.mainloop()
```

Root represents main program  
window; if not declared explicitly,  
it's the default

```
from Tkinter import *
Label(text='Hello GUI world!').pack()
mainloop()
```

The point here is that, some widget  
methods are exported as functions,  
allowing the code to be just 3 lines

# Basic Template

- `from Tkinter import *`
- `root = Tk()`
- Instantiate other widgets, attaching them to one another
- Configure the widgets
- Call the packer or grid for layout handling
- `root.mainloop()`

```
from Tkinter import *  
  
root = Tk()  
  
widget1 = Label(root)  
widget1.config(text='Hello world!')  
widget1.pack(side=TOP)  
  
widget2 = Button(root)  
widget2.config(text='Hello')  
widget2.pack(side=LEFT)  
  
widget3 = Button(root)  
widget3.config(text='Quit')  
widget3.pack(side=RIGHT)  
  
root.mainloop()
```



```
from Tkinter import *  
  
root = Tk()  
  
Label(root,text='Hello world!').pack(side=TOP)  
Button(root,text='Hello').pack(side=LEFT)  
Button(root,text='Quit').pack(side=RIGHT)  
  
root.mainloop()
```

# Extension to basic template

- Basic template will need to be modified to accommodate
  - Callback handlers (e.g. the action upon pressing button)
  - OOP methods – wrapping GUI into a class which subclasses the frame widget and the use of mixins
  - Etc.

# Callback handlers

- Refers to the code that will run when some widget is activated (e.g. button pressed)

- Example:

```
from Tkinter import *

def hello():
    print 'Hello'

def quit():
    print 'Bye'
    import sys; sys.exit()

root = Tk()

widget1 = Label(root)
widget1.config(text='Hello world!')
widget1.pack(side=TOP)

widget2 = Button(root)
widget2.config(text='Hello', command=hello)
widget2.pack(side=LEFT)

widget3 = Button(root)
widget3.config(text='Quit', command=quit)
widget3.pack(side=RIGHT)

root.mainloop()
```

This is one of several kinds of Tkinter callback protocols

The most general is widget bind methods



# Binding events with the bind method

```
from Tkinter import *

# event object contains information about the triggering event
def hello(event):
    print 'Hello'
    print 'Event info: Widget %s X=%s Y=%s' % (event.widget, event.x, event.y)

def quit(event):
    print 'Bye'
    import sys; sys.exit()

root = Tk()

widget1 = Label(root)
widget1.config(text='Hello world!')
widget1.pack(side=TOP)

widget2 = Button(root)
widget2.config(text='Hello')
widget2.bind('<Button-1>', hello)
widget2.pack(side=LEFT)

widget3 = Button(root)
widget3.config(text='Quit')
widget3.bind('<Button-1>', quit)
widget3.pack(side=RIGHT)

root.mainloop()
```

# Packing into a class

```
from Tkinter import *

class Hello(Frame):          # this GUI is an extended Frame
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        self.make_widgets()    # attach widgets to self

    def make_widgets(self):
        widget1 = Label(self)
        widget1.config(text='Hello world!')
        widget1.pack(side=TOP)

        widget2 = Button(self)
        widget2.config(text='Hello')
        widget2.bind('<Button-1>', self.hello)
        widget2.pack(side=LEFT)

        widget3 = Button(self)
        widget3.config(text='Quit')
        widget3.bind('<Button-1>', self.quit)
        widget3.pack(side=RIGHT)

    def hello(self, event):
        print 'Hello'
        print 'Event info: Widget %s X=%s Y=%s' % (event.widget, event.x, event.y)

    def quit(self, event):
        print 'Bye'
        import sys; sys.exit()

if __name__ == '__main__':
    Hello().mainloop()
```

Notice the difference in GUI  
See next slide for verification that  
it's because the buttons and label  
are attached to Frame rather than  
Tk

The present GUI is build from  
the Frame widget – larger GUIs  
are often built from Frame



# Attaching to Frame without using classes

```
from Tkinter import *

# event object contains information about the triggering event
def hello(event):
    print 'Hello'
    print 'Event info: Widget %s X=%s Y=%s' % (event.widget, event.x, event.y)

def quit(event):
    print 'Bye'
    import sys; sys.exit()

root = Frame() ←
root.pack()

widget1 = Label(root)
widget1.config(text='Hello world!')
widget1.pack(side=TOP)

widget2 = Button(root)
widget2.config(text='Hello')
widget2.bind('<Button-1>', hello)
widget2.pack(side=LEFT)

widget3 = Button(root)
widget3.config(text='Quit')
widget3.bind('<Button-1>', quit)
widget3.pack(side=RIGHT)

root.mainloop()
```

# Advantages of packaging GUI into classes that extend Frame

- A class is able to naturally represent various aspects of a GUI
  - Widgets are added by attaching to self
  - Callback handlers are registered as bound methods of self
  - State information are captured by fields of self
  - GUI as a Frame sub-class admits easily reusability, customization by inheritance, and composition attachment

# Using a GUI class

```
from Tkinter import *

class Hello(Frame):          # this GUI is an extended Frame
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        self.make_widgets() # attach widgets to self

    def make_widgets(self):
        widget1 = Label(self)
        widget1.config(text='Hello world!')
        widget1.pack(side=TOP)

        widget2 = Button(self)
        widget2.config(text='Hello')
        widget2.bind('<Button-1>', self.hello)
        widget2.pack(side=LEFT)

        widget3 = Button(self)
        widget3.config(text='Quit')
        widget3.bind('<Button-1>', self.quit)
        widget3.pack(side=RIGHT)

    def hello(self, event):
        print 'Hello'
        print 'Event info: Widget %s X=%s Y=%s' % (event.widget, event.x, event.y)

    def quit(self, event):
        print 'Bye'
        import sys; sys.exit()

if __name__ == '__main__':
    parent = Frame(None)      # make a container widget
    parent.pack()
    Hello(parent).pack(side=RIGHT)
    Button(parent, text='Attach', command=exit).pack(side=LEFT)
    parent.mainloop()
```

# Customizing/extending/inheriting a GUI class

```
from Tkinter import *

class Hello(Frame):
    # this GUI is an extended Frame
    def __init__(self, parent=None):
        Frame.__init__(self, parent)
        self.pack()
        self.make_widgets()
        # attach widgets to self

    def make_widgets(self):
        widget1 = Label(self)
        widget1.config(text='Hello world!')
        widget1.pack(side=TOP)

        widget2 = Button(self)
        widget2.config(text='Hello')
        widget2.bind('<Button-1>', self.hello)
        widget2.pack(side=LEFT)

        widget3 = Button(self)
        widget3.config(text='Quit')
        widget3.bind('<Button-1>', self.quit)
        widget3.pack(side=RIGHT)

    def hello(self, event):
        print 'Hello'
        print 'Event info: Widget %s X=%s Y=%s' % (event.widget, event.x, event.y)

    def quit(self, event):
        print 'Bye'
        import sys; sys.exit()

# to extend, need to override methods of Hello
# init of Hello is automatically called - which calls the present make_widgets
class HelloExtender(Hello):
    def make_widgets(self):
        Hello.make_widgets(self)
        widget = Button(self)
        widget.config(text='Extend')
        widget.bind('<Button-1>', self.quit)
        widget.pack(side=RIGHT)

    def hello(self, event):
        print 'Hello1'
        print 'Event info: Widget %s X=%s Y=%s' % (event.widget, event.x, event.y)

    def quit(self, event):
        print 'Bye1'
        import sys; sys.exit()

if __name__ == '__main__':
    HelloExtender().mainloop()
```

# Attaching Frames into single GUI (composition)

## a1.py

```
from Tkinter import *
```

```
class Demo(Frame):
```

```
    def __init__(self, parent=None):
```

```
        Frame.__init__(self, parent)
```

```
        self.pack()
```

```
        self.make_widgets()
```

```
    def make_widgets(self):
```

```
        widget = Label(self)
```

```
        widget.config(text='Hello 1')
```

```
        widget.pack(side=TOP)
```

## a2.py

```
from Tkinter import *
```

```
class Demo(Frame):
```

```
    def __init__(self, parent=None):
```

```
        Frame.__init__(self, parent)
```

```
        self.pack()
```

```
        self.make_widgets()
```

```
    def make_widgets(self):
```

```
        widget = Label(self)
```

```
        widget.config(text='Hello 2')
```

```
        widget.pack(side=TOP)
```

## onn.py

```
from Tkinter import *
```

```
demoModules = ['a1', 'a2']
```

```
parts = []
```

```
def addComponents(root):
```

```
    for demo in demoModules:
```

```
        module = __import__(demo)
```

```
        part = module.Demo(root)
```

```
        part.config(bd=2, relief=GROOVE)
```

```
        part.pack(side=LEFT, fill=BOTH)
```

```
        parts.append(part)
```

```
root = Tk()
```

```
Label(root, text='Multiple Frame Demo', bg='white').pack()
```

```
addComponents(root)
```

```
mainloop()
```

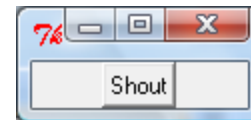
# Mixins

- Large GUI projects can be tedious
- We may implement common methods in a class and inherit them everywhere they are needed – such classes are called mixin classes
- The concept is almost like importing a module, but mixin classes can access the subject instance – self – to utilize per-instance state and inherited methods

Mixin

```
from Tkinter import *  
  
class GuiMixin:  
    def shout(self):  
        print 'Argh!'  
  
if __name__ == '__main__':  
    class TestMixin(GuiMixin, Frame):  
        def __init__(self, parent=None):  
            Frame.__init__(self, parent=None)  
            self.pack()  
            Button(self, text='Shout', command=self.shout).pack(fill=X)  
  
    TestMixin().mainloop()
```

Using mixin



# Appendix: Tkinter Widgets

# Widget classes

- Label
- Button
- Frame
  - Container for attaching/arranging other widgets
- Toplevel, Tk
  - A new window
- Message
  - Multiline label
- Entry
  - Single-line text-entry field
- Text
- Checkbutton
  - Like Radiobutton, but for multiple-choice selections
- Radiobutton
- Scale
- PhotoImage
- BitmapImage
- Menu
- Menubutton
  - Opens a menu
- Scrollbar
- Listbox
- Canvas
  - Graphic drawing area

# Widget: Label

```
from Tkinter import *  
  
root = Tk()  
Label(root, text='Multiple Frame Demo').pack()  
mainloop()
```



# Widget: Button

```
from Tkinter import *  
  
root = Tk()  
Button(root, text='Demo').pack()  
mainloop()
```



# Widget: Toplevel

- Tkinter GUIs always have a root window, regardless whether Tk() is called explicitly or not
- Any number of independent windows may be created by calling the Toplevel object
- But note that really there is only one process running

```
from Tkinter import *
```

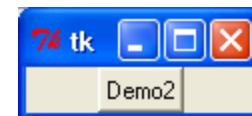
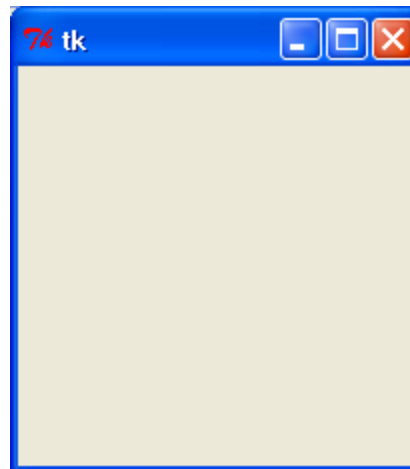
```
win1 = Toplevel()
```

```
win2 = Toplevel()
```

```
Button(win1, text='Demo1').pack()
```

```
Button(win2, text='Demo2').pack()
```

```
mainloop()
```



# Widget: Toplevel

Same meaning as:

```
from Tkinter import *
```

```
root = Tk()
```

```
win1 = Toplevel(root)
```

```
win2 = Toplevel(root)
```

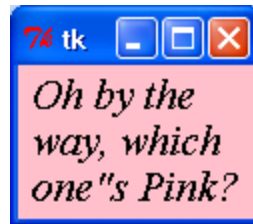
```
Button(win1, text='Demo1').pack()
```

```
Button(win2, text='Demo2').pack()
```

```
mainloop()
```

# Widget: Message

```
from Tkinter import *  
  
msg = Message(text='Oh by the way, which one"s Pink?')  
msg.config(bg='pink', font=('times', 16, 'italic'))  
msg.pack()  
mainloop()
```



# Widget: Entry

```
from Tkinter import *  
  
Entry(text='Demo').pack()  
mainloop()
```



# Widget: Text

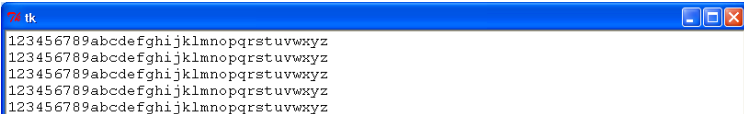
- The text widget is used to display text documents, containing either plain text or formatted text (using different fonts, embedded images, and other embellishments). The text widget can also be used as a text editor.

```
from Tkinter import *
```

```
root = Tk()  
text = Text(root, font=("Courier"))  
text.pack()
```

```
i='123456789abcdefghijklmnopqrstuvwxyz\n'  
for x in range(5):  
    text.insert(END, i)
```

```
mainloop()
```



```
tk  
123456789abcdefghijklmnopqrstuvwxyz  
123456789abcdefghijklmnopqrstuvwxyz  
123456789abcdefghijklmnopqrstuvwxyz  
123456789abcdefghijklmnopqrstuvwxyz  
123456789abcdefghijklmnopqrstuvwxyz
```

# Widget: Checkbutton

```
from Tkinter import *
```

```
Checkbutton(text='Demo').pack()  
mainloop()
```



# Widget: Radiobutton

```
from Tkinter import *  
  
Radiobutton(text='Demo').pack()  
mainloop()
```



# Widget: Scale

```
from Tkinter import *
```

```
w = Scale(from_=0, to=100)  
w.pack()
```

```
mainloop()
```



# Widget: Photoimage

```
from Tkinter import *  
  
root = Tk()  
igm = PhotoImage(file=r"c:\flowercup.gif")  
Button(root, image=igm).pack()  
  
root.mainloop()
```



# Widget: Menu

```
from Tkinter import *
```

```
root = Tk()
```

```
def hello():  
    print "hello!"
```

```
# create a toplevel menu
```

```
menubar = Menu(root)
```

```
menubar.add_command(label="Hello!", command=hello)
```

```
menubar.add_command(label="Quit!", command=root.quit)
```

```
# display the menu
```

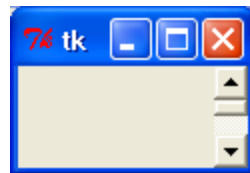
```
root.config(menu=menubar)
```

```
root.mainloop()
```



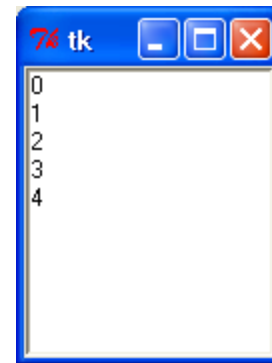
# Widget: Scrollbar

```
from Tkinter import *  
  
root = Tk()  
  
scrollbar = Scrollbar(root)  
scrollbar.pack(side=RIGHT)  
  
mainloop()
```



# Widget: Listbox

```
from Tkinter import *  
  
root = Tk()  
  
listbox = Listbox(root)  
listbox.pack()  
  
for i in range(5):  
    listbox.insert(END, i)  
  
mainloop()
```



# Widget: Canvas

- The canvas is a general purpose widget, which is typically used to display and edit graphs and other drawings.

```
from Tkinter import *
```

```
master = Tk()
```

```
w = Canvas(master, width=200, height=100)
```

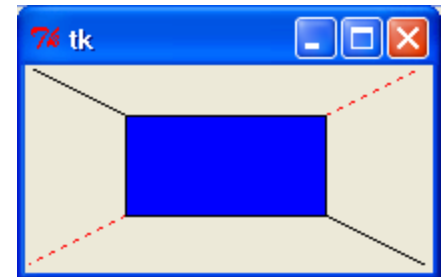
```
w.pack()
```

```
w.create_line(0, 0, 200, 100)
```

```
w.create_line(0, 100, 200, 0, fill="red", dash=(4, 4))
```

```
w.create_rectangle(50, 25, 150, 75, fill="blue")
```

```
mainloop()
```



# **Technical Aside: Python Class Mechanism**

# Python is an OOP language

- There is no standard specification for what makes a programming language an OOP language
- But you know one when you see one
- The name OOP itself suggest the direction of thought in this regard – OBJECT-ORIENTED
- Thus an OOP language is one that makes some fuss over objects
- C++ is regarded as an OOP language because it implements classes which may be instantiated into objects
- Python goes further than that in that everything in Python is an object, with more complex objects deriving from more basic ones

# Python Object

- Every object in Python comprises:
  - An identity
  - A value (its attributes)
  - A type
  - One or more bases

```
>>> two = 2 ❶ ← name
>>> type(two)
<type 'int'> ❷ ← type
>>> type(type(two))
<type 'type'> ❸ ← type itself is an object
>>> type(two).__bases__
(<type 'object'>,) ❹ ← value/attributes
>>> dir(two) ❺ ←
['_abs_', '_add_', '_and_', '_class_', '_cmp_', '_coerce_',
'_delattr_', '_div_', '_divmod_', '_doc_', '_float_',
'_floordiv_', '_format_', '_getattr_', '_getnewargs_',
'_hash_', '_hex_', '_index_', '_init_', '_int_', '_invert_',
'_long_', '_lshift_', '_mod_', '_mul_', '_neg_', '_new_',
'_nonzero_', '_oct_', '_or_', '_pos_', '_pow_', '_radd_',
'_rand_', '_rdiv_', '_rdivmod_', '_reduce_', '_reduce_ex_',
'_repr_', '_rfloordiv_', '_rshift_', '_rmod_', '_rmul_',
'_ror_', '_rpow_', '_rrshift_', '_rshift_', '_rsub_',
'_rtruediv_', '_rxor_', '_setattr_', '_sizeof_', '_str_',
'_sub_', '_subclasshook_', '_truediv_', '_trunc_', '_xor_',
'conjugate', 'denominator', 'imag', 'numerator', 'real']
```

# Python Object Model

- There are two kinds of objects in Python:
  1. *Type objects* - can create instances, can be subclassed.
  2. *Non-type objects* - cannot create instances, cannot be subclassed.
- `<type 'type'>` and `<type 'object'>` are two primitive objects of the system.
- `objectname.__class__` exists for every object and points the type of the object.
- `objectname.__bases__` exists for every type object and points the superclasses of the object. It is empty only for `<type 'object'>`.
- To create a new object using subclassing, we use the `class` statement and specify the bases (and, optionally, the type) of the new object. This always creates a type object.
- To create a new object using instantiation, we use the call operator `()` on the type object we want to use. This may create a type or a non-type object, depending on which type object was used.
- Some non-type objects can be created using special Python syntax. For example, `[1, 2, 3]` creates an instance of `<type 'list'>`.
- Internally, Python *always* uses a type object to create a new object. The new object created is an instance of the type object used. Python determines the type object from a `class` statement by looking at the bases specified, and finding their types.
- `issubclass(A, B)` (testing for superclass-subclass relationship) returns `True` iff:
  1. `B` is in `A.__bases__`, OR
  2. `issubclass(Z, B)` is true for any `z` in `A.__bases__`.
- `isinstance(A, B)` (testing for type-instance relationship) returns `True` iff:
  1. `B` is `A.__class__`, OR
  2. `issubclass(A.__class__, B)` is true.

# Exceptions

# Exceptions

- Exceptions refer to errors, or more precisely, a pattern of code behaviour that is not what is planned by the programmer. Exception handling refers to ways of programmatically handling such deviations from wanted behaviours.
- The statements involved in this topic are:
  - try/except/else/finally
    - Catch and recover from exceptions
  - raise
    - Trigger exception manually

# Examples

```
# try-except-else-finally
# basic: try-except, try-finally
try:
    val = 1
    print 'statements'
    # 1 / 0
    raise SyntaxError, 'abc'      # SyntaxError is raised, data part is optional
except NameError:
    print 'NameError'
except (AttributeError, TypeError):
    print 'One of several errors'
except SyntaxError, extradata:
    print 'SyntaxError'
    print 'Value obtained:', extradata
except:
    print 'All other errors'
else:
    print 'No error occurred'
finally:
    print 'This will always run'
```

# A useful idiom

- It is good to wrap statements which may fail in a try-except block
  - E.g. a function `getDataFromYahooFinance()`
- Here is a code that will keep executing the unreliable function until it succeeds:

```
while 1:  
    try:  
        getDataFromYahooFinance()  
        break  
    except:  
        pass
```

# What's an exception, really?

- Do this and have a look:
  - `import exceptions`
  - `help(exceptions)`
- Exceptions are either string-based or class-based...class-based is the standard
- The Python built-in exception classes
  - Exception
    - Top-level superclass of exceptions
  - StandardError
    - Superclass of all built-in error exceptions
  - ArithmeticError
    - Superclass of all numeric errors
  - Etc.

# Example: custom exception

```
class General(Exception):
    def __init__(self,line):
        self.line = line
    def __str__(self):
        return "Sorry my mistake"

def raiser():
    raise General(12345)    # this goes into the line variable

try:
    raiser()
except General, data:
    import sys
    print 'caught:', sys.exc_info()[0]
    print 'Data: ', data.line
    raise    # reraise the most recent exception
```

## OUTPUT:

Traceback (most recent call last):

```
File "C:\Documents and Settings\chonghuitan\My Documents\NetBeansProjects\NewPythonProject\src\try.py", line 11, in <module>
    raiser()
File "C:\Documents and Settings\chonghuitan\My Documents\NetBeansProjects\NewPythonProject\src\try.py", line 8, in raiser
    raise General(12345)    # this goes into the line variable
__main__.General: Sorry my mistake
caught: <class '__main__.General'>
Data: 12345
```

# Exercises

# Exercises

- 1) In the idiom involving try-except statement block created to try to execute an unreliable statement until it succeeds.....improve upon the idiom by getting it to try to execute the unreliable statement for a certain number of times before giving up
- 2) Illustrate how to overload the operator ==
- 3) In the context of Python classes, what does self mean?

# Exercises

- 4) Why is the first argument in a class method function special?
- 5) Which operator overloading method is the most commonly used?
- 6) How can you augment, instead of completely replace, an inherited method?
- 7) Explain what are string-based exceptions and give an example.

# Exercises

- 8) Explain how you can specify the error message text in class-based exceptions
- 9) Use classes to model polynomials
- 10) Use classes to solve the problem of easily converting between Kelvin, Celsius, Fahrenheit and Rankine scales of temperature
- 11) Use Tkinter to allow users to create an EquityIndex object via GUI rather than by Python programming

# References

## Ref.

- Part VI. Functions, from the book: Learning Python by Mark Lutz
- Part VII. Modules, from the book: Learning Python by Mark Lutz
- Operator Overloading
  - <http://docs.python.org/reference/datamodel.html>

# Ref.

- Python Exceptions
  - <http://docs.python.org/tutorial/errors.html>
  - <http://docs.python.org/library/exceptions.html>
- Python Types and Objects
  - [http://www.cafepy.com/article/python\\_types\\_and\\_objects/python\\_types\\_and\\_objects.html](http://www.cafepy.com/article/python_types_and_objects/python_types_and_objects.html)
  - <http://www.python.org/download/releases/2.2.3/descrintro/>