Art of Python

Benny Khoo © 2014

Course objectives

- Try to present Python from creator perspective e.g. motivation
- Opportunity to sharpen your skillset –
  - you can still always Google for details (e.g. syntax, documentation) but it is not obvious how to Google to develop your skills
- Teach you how to think so that you can express your creativity through Python
  - Python is the medium of your expression but you are still the main artist

Course contents

Object oriented system

Functional programming

Generic programming

Some design patterns

Some algorithms

Some data structure

Some software principals

Motivations

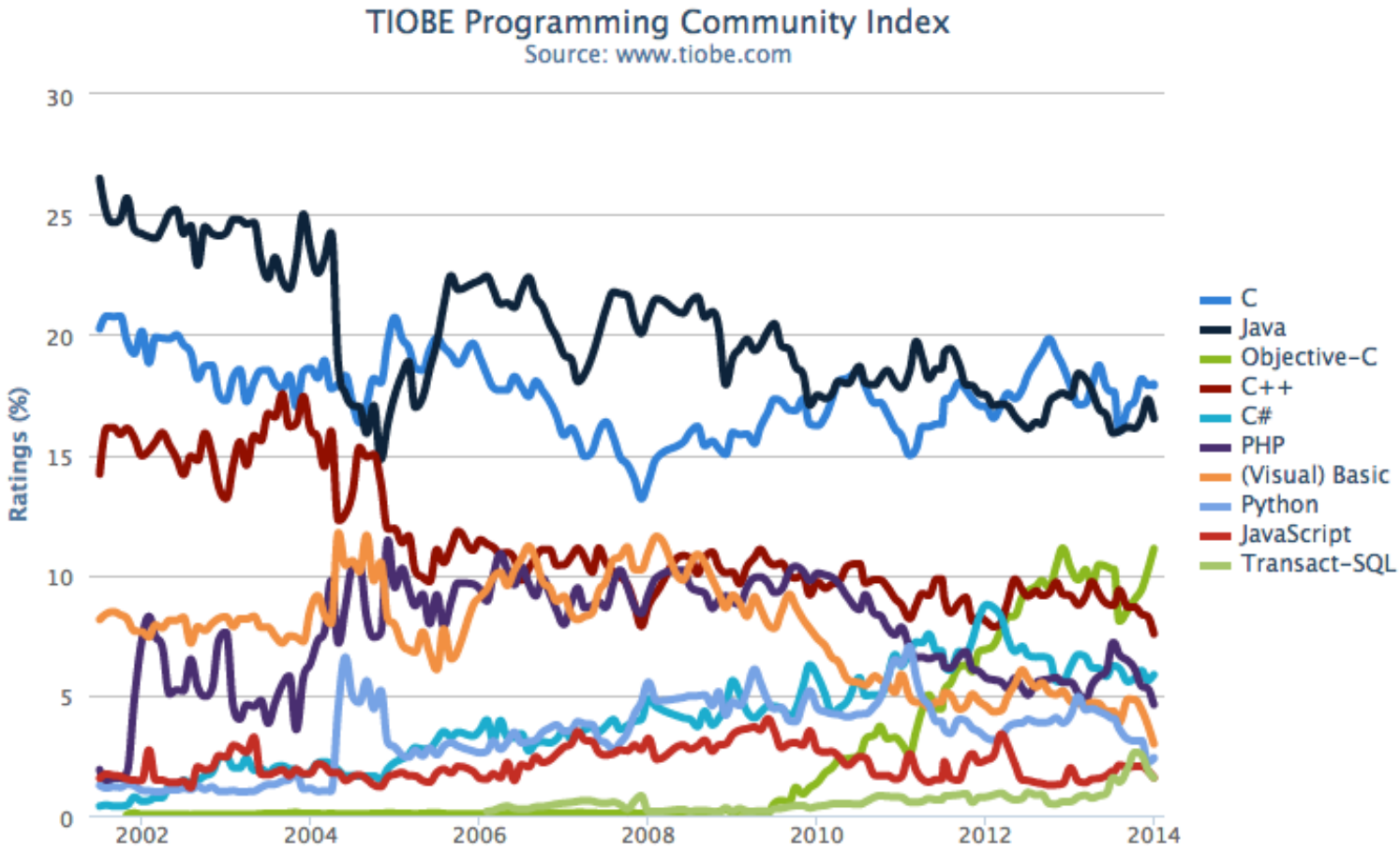
- www server side programming
  - Django, Google App Engine
- Scientific programming
  - Numpy, SciPy, ipython
- Data mining
  - NLTK (Natural Language Toolkit), Encog (AI framework)
- Native user interface
  - PyQt, wxPython, PyGtk, Tkinter
- EDA
  - PyCell, Helix, OpenEDA

Tiobe Popularity Contest

2014	2013	Language	Ratings	Change
1	1	C	17.871%	+0.02%
2	2	Java	16.499%	-0.92%
3	3	Objective-C	11.098%	+0.82%
4	4	C++	7.548%	-1.59%
5	5	C#	5.855%	-0.34%
6	6	PHP	4.627%	-0.92%
7	7	(Visual) Basic	2.989%	-1.76%
8	8	Python	2.400%	-1.77%
9	10	JavaScript	1.569%	-0.41%
10	22	Transact-SQL	1.559%	+0.98%
11	12	Visual Basic .NET	1.558%	+0.52%
12	11	Ruby	1.082%	-0.69%
13	9	Perl	0.917%	-1.35%
14	14	Pascal	0.780%	-0.15%
15	17	MATLAB	0.776%	+0.14%

Source www.tiobe.com

Tiobe Popularity Contest - Contnue



Zen of Python – PEP 20

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Python 123

- Do you still use Python 2?
- Use 2to3 program to convert your python 2 script to python 3

Resources

- Pydoc
- http://docs.python.org/3.3/tutorial/

Interactively

% python –i <script>
% python –mpdb <script>
% python

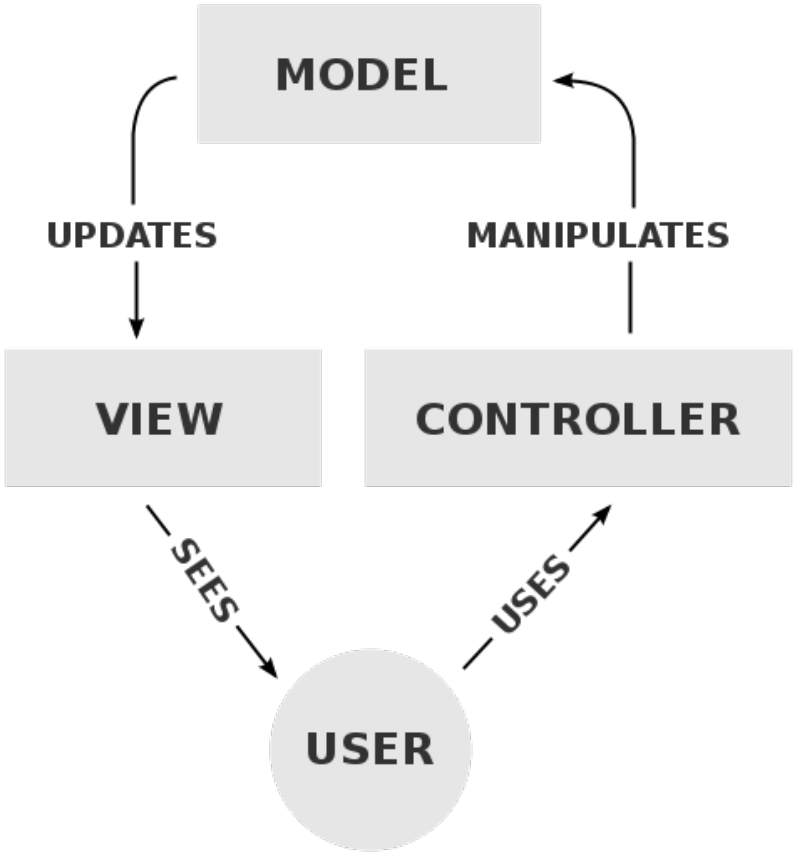
Debugging

- Demo how to use python for debugging

IDE

- WingIDE, PyCharm
- Hyperlinks
- Code folding
- GUI debugger
- Quick help
- Automatic code refactoring
- Automatic code completion
- Auto such and such …

Model View Controller (MVC)



- Primary use in UI design
- It can be a useful model to understand any practical applications

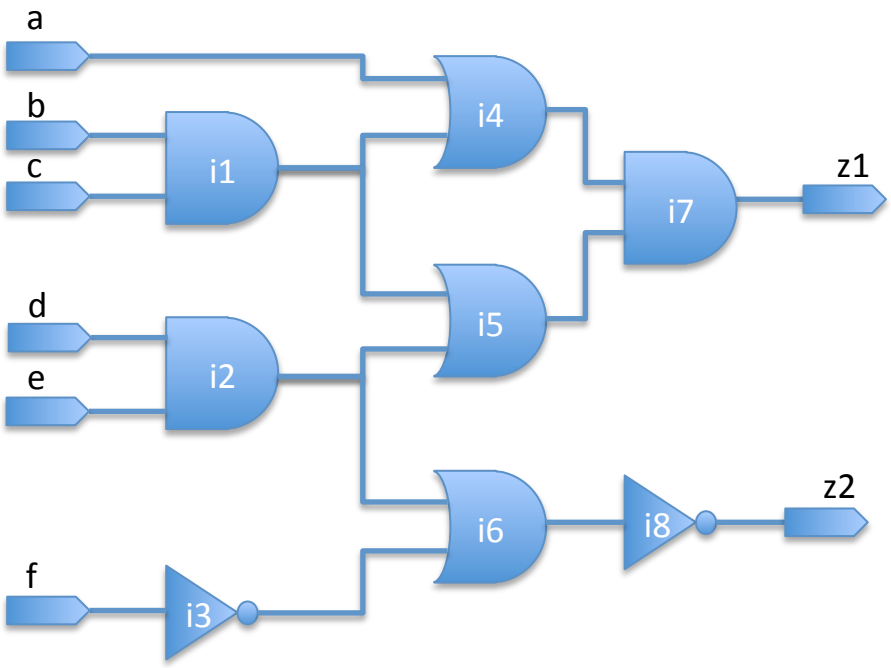
Design Patterns

- The study of software patterns used to solve commonly reoccurring problem
- Made famous by the GoF book
- For us – useful to study OO system pratically
- Haven't seen it yet? Go check it out here.

Anti-pattern

- Documented study of commonly re-occur issues that are usually ineffective and counter-productive in software projects
  - may include non engineering issues such as project management issue, culture, ego etc.
- Interesting? Go check it out here.

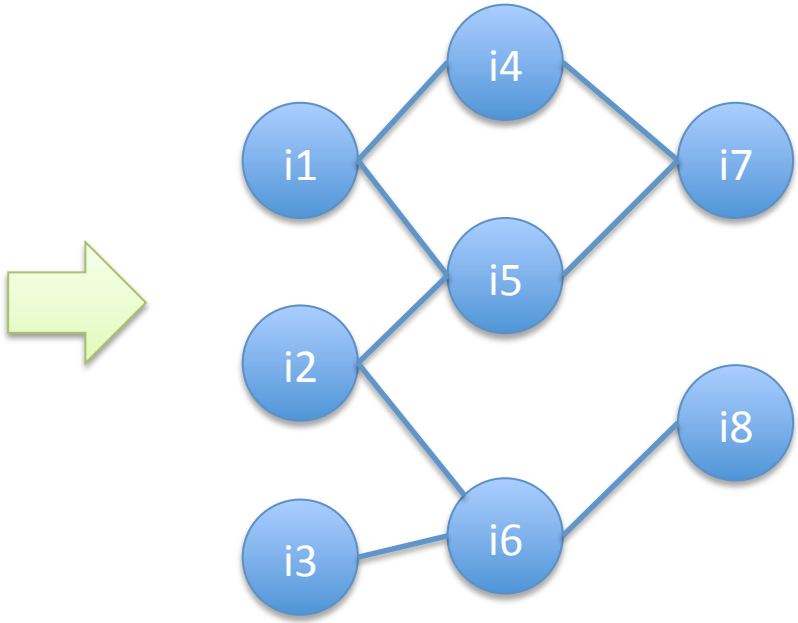
Circuit netlist




```
# transistor level
ckt and a b z
q1 vcc a z p
q2 vcc b z p
q3 vcc a n1 n
q4 n1 b z n
end
…

# gate level
ckt top a b c d e f z1 z2
i1 b c n1 and
i2 d e and
i3 f n3 not
i4 a n1 n4 or
i5 n1 n2 n5 or
i6 n2 n3 n6 or
i7 n4 n5 z1 and
i8 n6 z2 not
end
```

Circuit model

```
# transistor level
ckt and a b z
q1 vcc a z p
q2 vcc b z p
q3 vcc a n1 n
q4 n1 b z n
end
…

# gate level
ckt top a b c d e f z1 z2
i1 b c n1 and
i2 d e and
i3 f n3 not
i4 a n1 n4 or
i5 n1 n2 n5 or
i6 n2 n3 n6 or
i7 n4 n5 z1 and
i8 n6 z2 not
end
```



Prelude – parsing netlist file

* See example on separate file

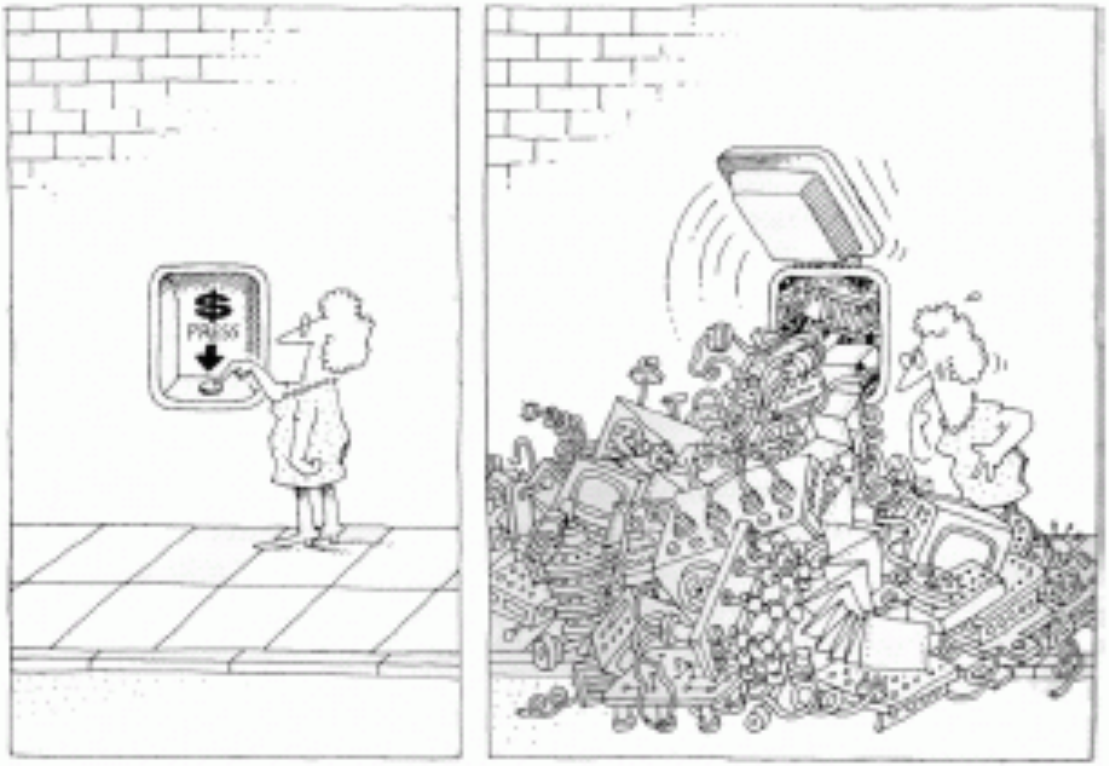
Understanding Chaos

1. Complexity takes the form of a hierarchy
2. Hierarchy are composed of identifiable parts
3. Intracomponent linkages are stronger than intercomponent linkages
4. Hierarchic systems are usually composed only few kinds in various combinations

The Object model

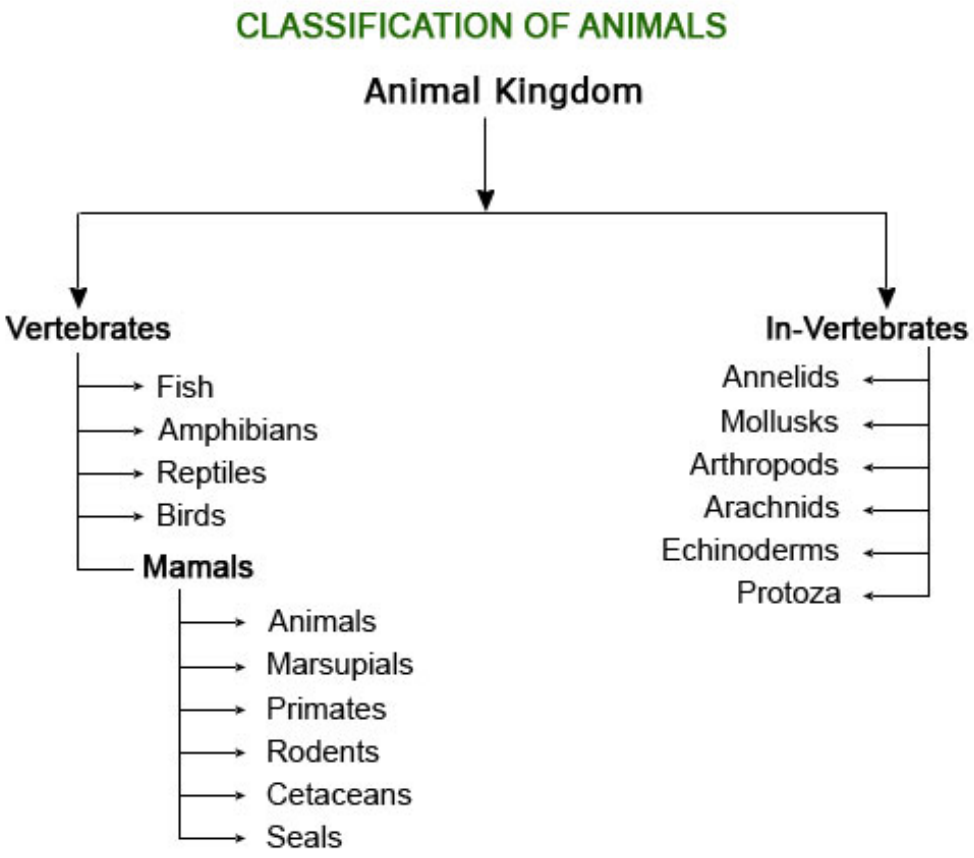
- Model inspired by physical system. Think: -
  - Structure of Plants and Animals, structure of matter, social institutions, human inventions
- Characteristics: -
  1. Abstraction
  2. Encapsulation
  3. Modularity
  4. Hierarchy
  5. Typing

Illusion of simplicity



The task of the software development team
is to engineer the illusion of simplicity.

How does object hierarchy looks like?



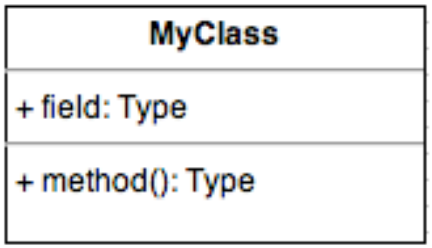
Object Oriented Analysis+Design

- Satisfies a given functional specification
- Decomposing a system to identifiable objects and how various objects are linked
- Identify patterns and protocols

The class statement

- Class definition

```
class MyClass:
    pass
```



- Becomes an object when instantiated

```
>>> a = MyClass()
>>> a
<__main__.MyClass object at 0x10cc08090>
```

- You may instantiate multiple objects as needed

```
>>> a = MyClass()
>>> b = MyClass()
```

Class structure

- Methods: add, remove
- Special methods: __init__, __del__
- Attributes: name, version
- Private attributes: _cell
- Class attribute: numOfCell

```
class Library:
    numOfCell = 0

    def __init__(self, name, version):
        self.name = name
        self.version = version

        # init a dict to keep all store entries
        self._cell = {}

        Library.numOfCell += 1

    def __del__(self):
        Library.numOfCell -= 1

    def add(self, cell, name):
        self._cell[name] = cell

    def remove(self, name):
        del self._cell[name]
```

Class instances

- Use the class name to construct object which in turn invokes __init__ internally
- Use the dot (.) operator to access object attributes and to invoke methods on object

```
# create a few libraries

# this invokes Library.__init__('ckt1.net', 1)
a = Library("ckt1.net", 1)
b = Library("ckt2.net", 3)


# accessing attributes

# prints 'ckt1.net' and 1
print(a.name, a.version)

# invoke some methods
a.add(someCell)
b.remove(cellName)
```

Scoping rules

- Use *self* to access attributes in the object
- Self tracks the current object of class under processing

```
(Pdb) p self
<__main__.Library object at 0x1100cd610>
(Pdb) p self.__dict__
{'_cell': {}, 'version': 1, 'name': 'ckt1.net'}
```

```
class Library:
    …
    def hasCell(self, name):
        return name in self._cell

    def add(self, cell, name):
# if hasCell(name) gets error
# NameError: global name 'hasCell' is not defined
        if self.hasCell(name): # This works!
            print("Warning: cell with name %s
already exists! replacing..." % name)
        self._cell[name] = cell

    def remove(self, name):
        if not self.hasCell(name):
            print("Error: can't find any cell name
with %s to remove" % name)
            return
        del self._cell[name]
```

Properties

```
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return 2*math.pi*self.radius**2

    @property
    def perimeter(self):
        return 2*math.pi*self.radius
```

```
>>> a = Circle(3)
>>> a.radius
3
>>> a.perimeter
18.84955592153876
>>> a.area
56.548667764616276
```

Enable methods to be accessed like natural attribute

Properties setter

```
class Unnameable:
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, value):
        assert isinstance(value, str)
        self.__name = value

    @name.deleter
    def name(self):
        raise TypeError("can't delete name")
```

```
>>> a = Unnameable('benny')
>>> a.name
'benny'
>>> a.name = "khoo"
>>> a.name
'khoo’
>>> del(a.name)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "property.py", line 32, in name
    raise TypeError("can't delete name")
TypeError: can't delete name
```

Privacy is important

- Privacy is important to indicate your design intention to the reader. It is NOT design for security
- All attributes are accessible out of class scope by default
- If you are liberal thinker – use one underscore prefix '_' to indicate your private intention
- For absolute privacy - use the double underscore '__' prefix to hide attributes from direct access
  - No special cryptography here if you try hard enough you can still figure it out

```
>>> a.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Unnameable' object has no attribute '__name'
```

Inheritance

- Allow code in different classes to share similar characteristics while allowing specialization and modification of behavior
- Enable creative assembly and architecture of classes hierarchy. Can produce more elegant and readable code
- Design inspired by nature
- Base class – the class at the root of inheritance hierarchy. Usually at abstract, conceptual level (e.g. Animal)
- Derived class – is the class that make derivation from base class. Usually at implementation, concrete level.

```
# base class
class Animal:
    def speak(self):
        raise NotImplementedError()

    def speak_twice(self):
        self.speak()
        self.speak()

# derived class of animal
class Dog(Animal):
    def speak(self):
        print("woff!”)

# derived class of animal
class Cat(Animal):
    def speak(self):
        print("meow”)
```

Dynamic binding

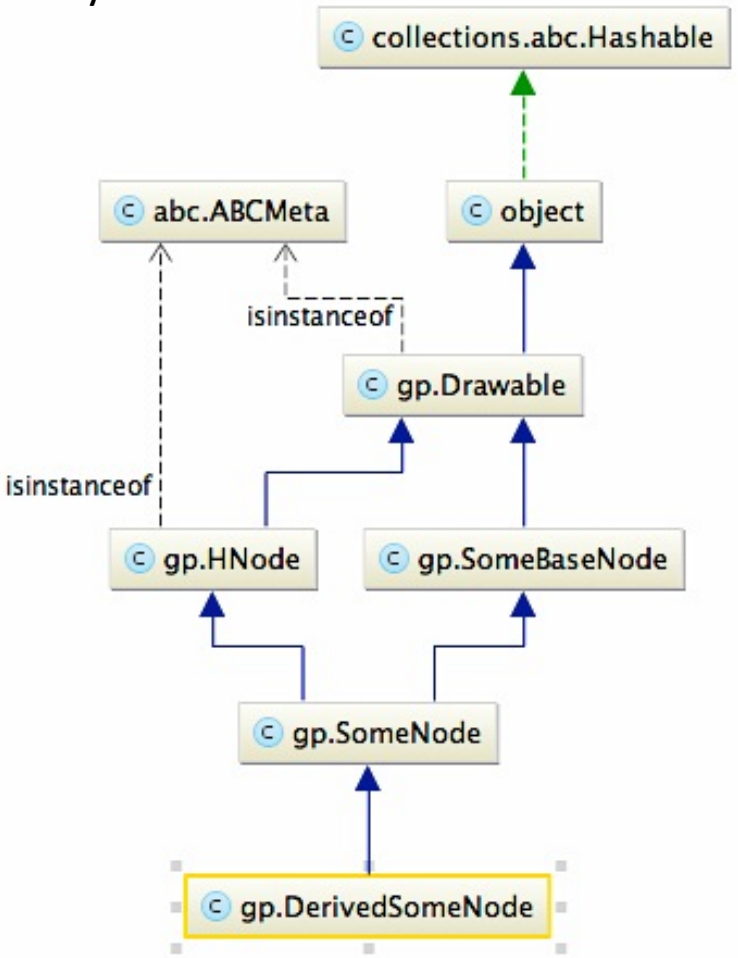
- Ability to invoke method or access attributes of the object independent from the type of the object
- In Python this can be done by locating the attribute within the instance itself, next searching through the inheritance hierarchy in order
- Thus accessing *obj.name* will works on any object that happens to have a name attribute
- Sometimes called “duck typing” – “if it looks like, quacks like, and walks like a duck, then it is a duck”

```
import random

def generateAnimal(n):
    result = []
    for x in range(n):
        cls = random.choice([Cat, Dog])
        result.append(cls())
    return result

for x in generateAnimal(3):
    x.speak_twice()
```

Identity crisis: who am I?

```
# test if instance a inherits B in inheritance hierarchy
isinstance(a, B)
⇒ True|False

# test if C inherits B in inheritance hierarchy
issubclass(C, B)
=> True|False
```

- Testing inheritance hierarchy
- Recommended for:
  - IDE auto-completion
  - Type checking
  - Improving code readability



Initializing derived class

- Use super() to invoke __init__ method defined in base class. Next initialize attributes in derived object in order
- In Python there is no restriction to do pre and post base class init. When unsure follow pre init.
- Use scatter operators * and ** to pass along arguments and keyword arguments to __init__ that participates in inheritance chain.
- Scatter operators are not mandatory but good practices

```
class Animal(object):
    def __init__(self, name, *args, **kwargs):
        self.name = name

class Dog(Animal):
    def __init__(self, *args, **kwargs):
# initialize base first
# same: Animal.__init__(*args)
        super().__init__(*args)
# initialize me
…
```

Modifying existing behaviors

- Use super() to augment existing method
- Useful in code refactoring since touching the base code can have wide impact
- You can apply this technique to modify behaviors of built-ins Python library too

```
class Animal(object):
…
    def showAffection(self):
        self.speak_twice()

class Cat(Animal):
…
  def showAffection(self):
  # apply pre-processing for purrr...meow! meow!
        print("purrr...")
        super().showAffection()

        # apply post processing
```

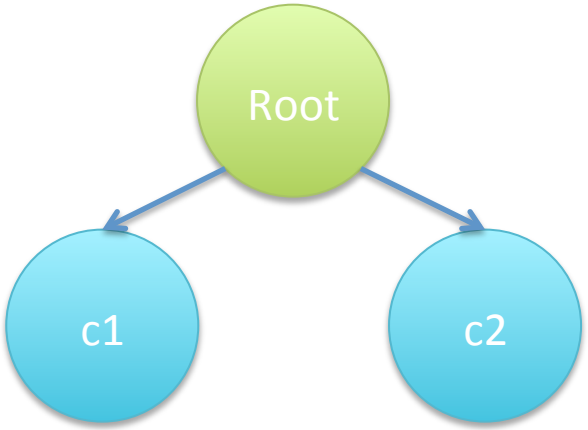
Practical use of inheritance

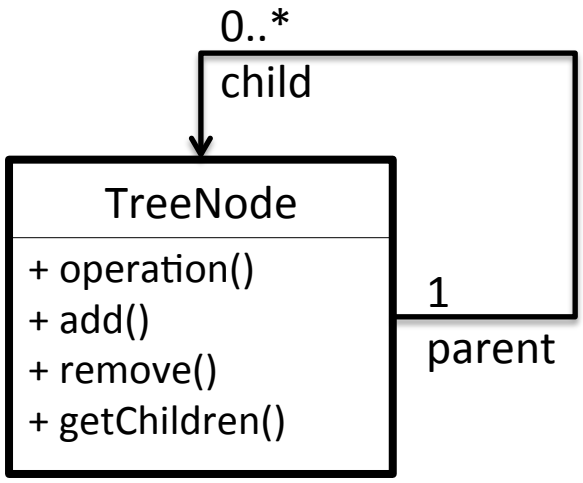
1. Your inheritance hierarchy represents an "is-a" relationship and not a "has-a" relationship.
2. You can reuse code from the base classes.
3. You need to add specialization to certain data-type.
4. The class hierarchy is reasonably shallow, and other developers are not likely to add many more levels.
5. You want to make global changes to derived classes by changing a base class.

http://msdn.microsoft.com/en-us/library/27db6csx(v=vs.90).aspx

Case study: building a tree

```
class TreeNode:
    def __init__(self, name):
        self._children = []
        self.name = name
        self.parent = None
        # init other data associated with the node

    def getChildren(self):
        return self._children

    def add(self, child):
        self._children.append(child)
        child.parent = self

    def dump(self, depth=0):
        print("%s%s" % (depth*'   ', self.name))
        for childNode in self._children:
            childNode.dump(depth=depth+1)
```



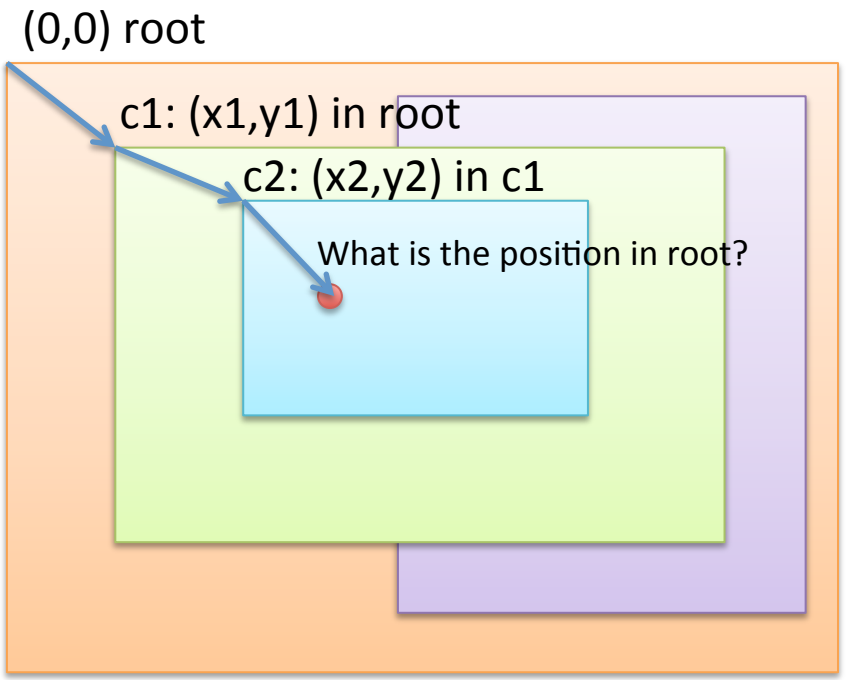


Case study: building a tree (2)

```
_id = 0
def buildTreeRandomly(parent, level=2, limit=5, depth=1):
    global _id
    for i in range(random.randint(1, limit)):
        _id += 1
        childNode = TreeNode('c%s' % _id)
        parent.add(childNode)

    if depth >= level:
        return

    for childNode in parent.getChildren():
        buildTreeRandomly(childNode, depth=depth+1, level=level, limit=limit)

root = TreeNode('root')
buildTreeRandomly(root, level=2)
root.dump()
```

Case study: coordinate conversion in window toolkit

- In typical window toolkit, it is common practice to track the origin position of a window as offset from its parent window



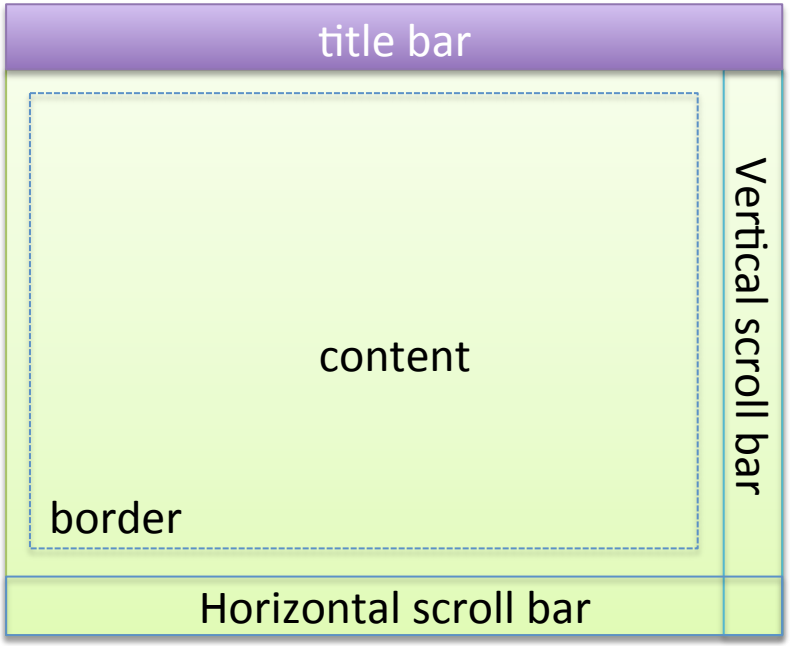
Case study: coordinate conversion in window toolkit

```
class Window:
    def __init__(self):
        self._children = []
        self.parent = None
        self.offset = 0

    def add(self, child):
        self._children.append(child)
        child.parent = self

    def convertToParentOffset(self, pos):
        raise NotImplementedError
```

```
class RootWindow(Window):
    def convertToParentOffset(self, pos):
        return pos

class ChildWindow(Window):
    def __init__(self, offset):
        super().__init__()
        self.offset = offset

    def convertToParentOffset(self, pos):
        return
self.parent.convertToParentOffset(self.offset) + pos
```

Case study: decorating window – problem statements

- How to decouple implementations in content from decorating implementations (title, vertical and horizontal scroll bar)
- How to assemble all possible combination of decoration features without touching the implementation: -
  - With border, title bar, no scroll bars
  - With vertical scroll bar only, no horizontal scroll bar, no title bar
  - The list goes on …



Case study: wrapping things with decoration

```
class VerticalScrollBarDecorator(WindowDecorator):
    def __init__(self, content):
        super().__init__(content)

    def drawVerticalScrollBar(self, depth=0):
        print("%sdraw vertical scroll bar" % ('   ' * depth))

    def draw(self, **kwargs):
        super().draw(**kwargs)
        self.drawVerticalScrollBar(**kwargs)
```

```
class HorizontalScrollBarDecorator(WindowDecorator):
    def __init__(self, content):
        super().__init__(content)

    def drawHorizontalScrollBar(self, depth=0):
        print("%sdraw horizontal scroll bar" % ('   ' * depth))

    def draw(self, **kwargs):
        super().draw(**kwargs)
        self.drawHorizontalScrollBar(**kwargs)
```

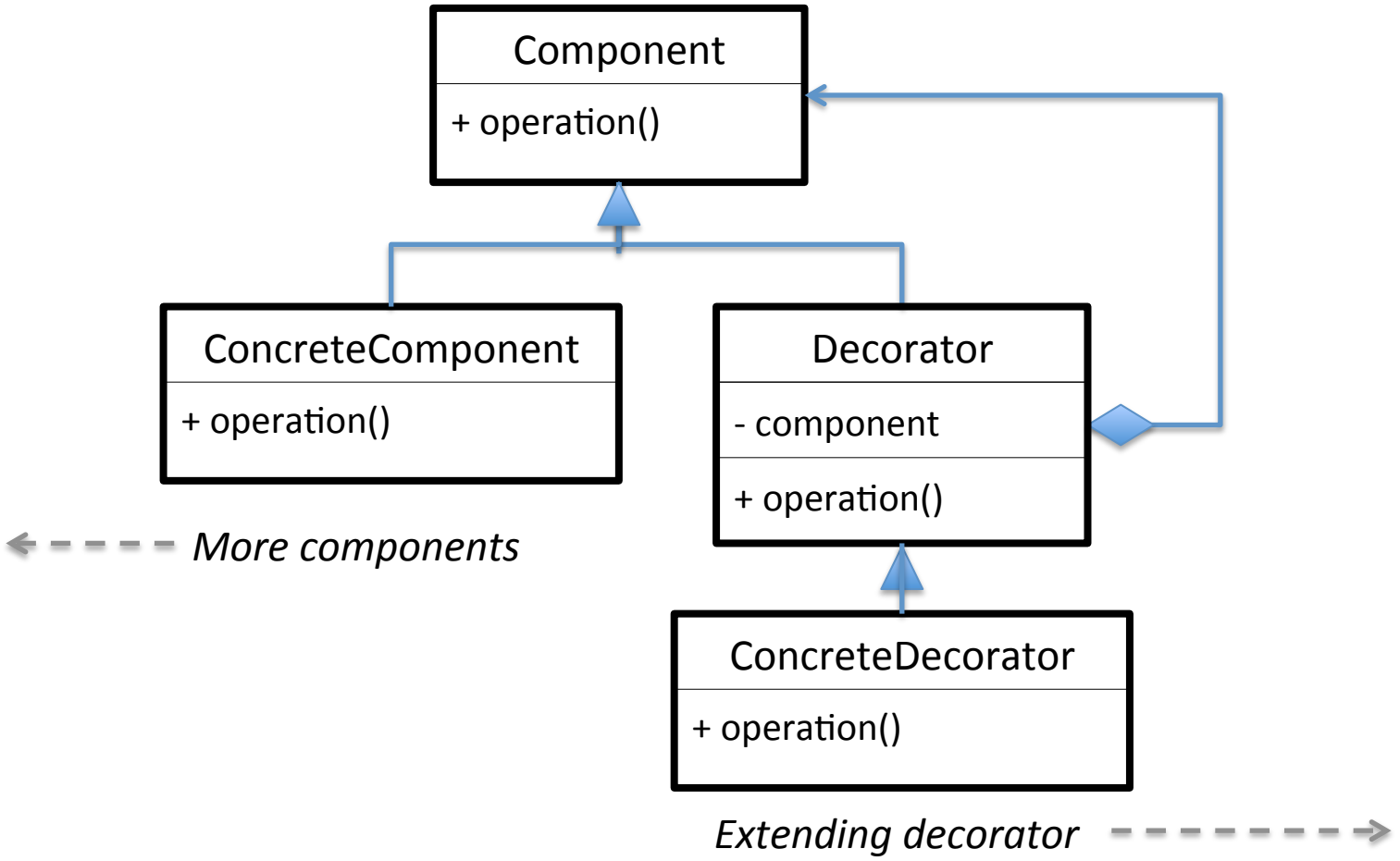
```
class WindowDecorator(Window):
    def __init__(self, content):
        self.content = content

    def draw(self, depth=0, **kwargs):
        print("%s{draw decorated content" % ('   ' * depth))
        self.content.draw(depth=depth+1, **kwargs)
        print("%s}" % ('   ' * depth))
```

Case study: decorating window – in action

```
decoratedWindow = HorizontalScrollBarDecorator(
      VerticalScrollBarDecorator(ContentWindow()
      )
   )
decoratedWindow.draw()
```

```
 {draw decorated content
   {draw decorated content
      draw content window
   }
   draw vertical scroll bar
}
draw horizontal scroll bar
```

Decorator pattern – UML diagram

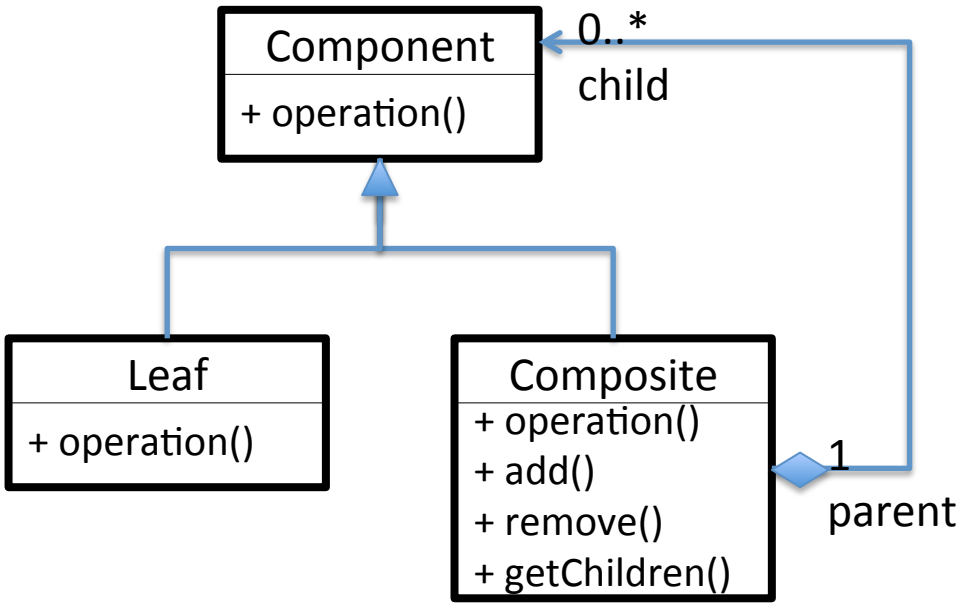


Decorator pattern – features in nutshell

- Alternative to sub-classing (adding additional behaviors) you may choose to decouple functions to a separate object (content)
- Features: -
  - Loose coupling between decorating and concrete implementations (via base class only) allowing functions to evolve separately
  - Permutations of combination are made possible via cooperating decorator classes and can be assembled at ease

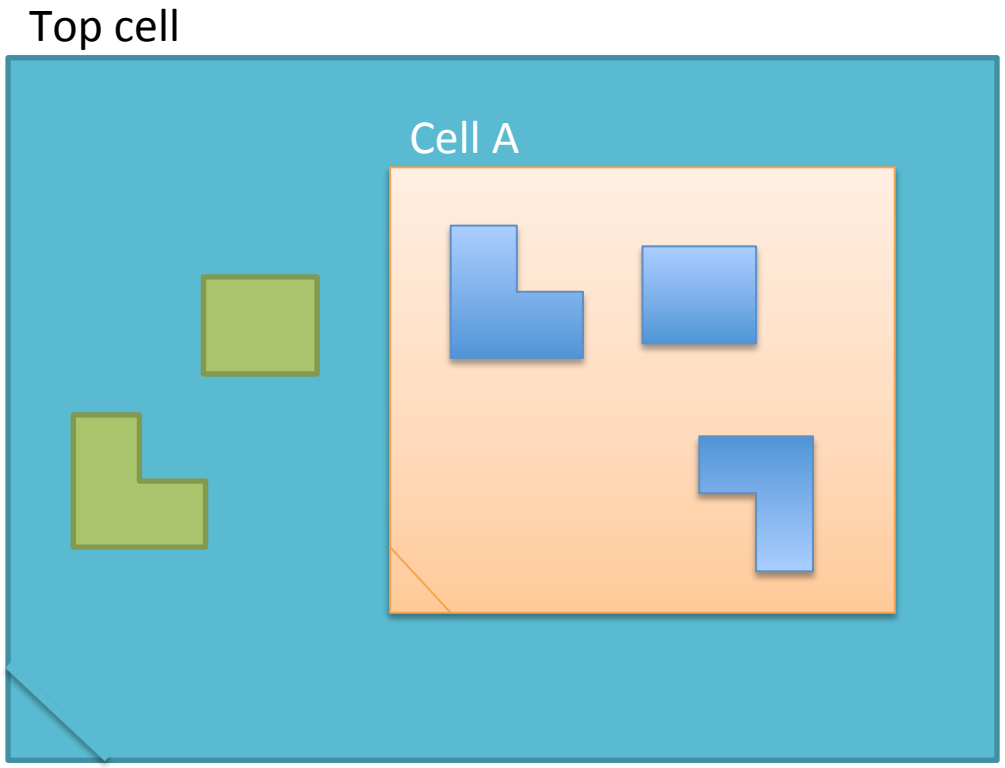
Recursive composition pattern

- A composite is an object that assembles objects of its kind (think group)
- Manipulating a group of object is treated as if you would manipulate a single object
- By treating composite and other leaf objects similarly we can reduce complexity resulting in less error prone code

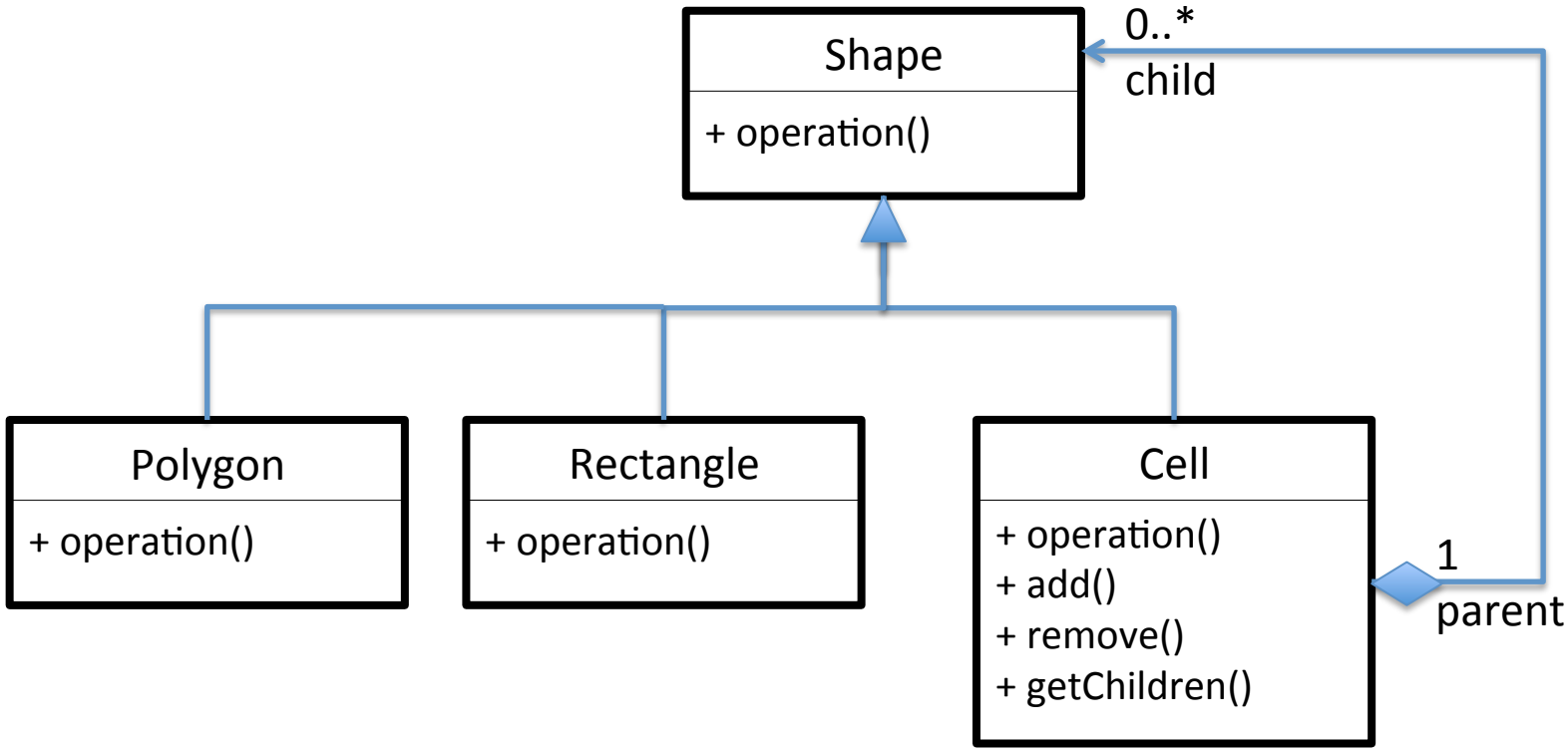


Recursive composition in CAD system

- Cell groups a list of shapes. Cell is the composite object participating in recursive composition
- Cell is in turn assemble-able inside another cell in a parent-child relationship
- Shapes in cell can be manipulated at once as if you would manipulate a single shape



Recursive composition in CAD system – UML diagram



Exercises for OO Design

- Based on the ckt example, complete the formulation for building a ckt data model using OO architecture. Consider the following issues: -
  - How to represent connectivity?
  - How to represent instantiations?
  - How to represent hierarchy of instances?
  - How to represent collection of cells?

  (Hint: use recursive composition, inheritance whenever is possible.)
- Layout your initial OO plan using Python.

Graph

Case study: building graph

```
class GEdge:
    def __init__(self, weight=0):
        self._inNode = None
        self._outNode = None
        self.weight = weight

    def connect(self, inNode, outNode):
        self._inNode = inNode
        self._outNode = outNode
        outNode._inEdges.append(self)
        inNode._outEdges.append(self)

    def __str__(self):
        # for debugging print
        return "%s -> %s" % (self._inNode.name, self._outNode.name)
```

```
class GNode:
    def __init__(self, name):
        self.name = name
        self._outEdges = []
        self._inEdges = []
```

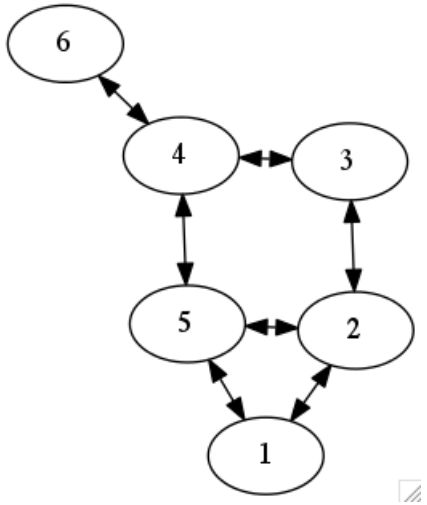
Case study: building graph (2)

```
class Graph:
    def __init__(self):
        self._nodes = {}
        self._edges = []

    def addNode(self, label):
        n = GNode(label)
        self._nodes[label] = n
        return n

    def addEdge(self):
        e = GEdge()
        self._edges.append(e)
        return e
```

```
# define some accessor
    def hasNode(self, name):
        return name in self._nodes

    def getNode(self, name):
        return self._nodes[name]

    def getNodes(self):
        return self._nodes.values()
```

Case study: building graph (3)

```
data = {'1': ['2', '5'],
        '2': ['1', '3', '5'],
        '3': ['2', '4'],
        '4': ['3', '5', '6'],
        '5': ['1', '2', '4'],
        '6': ['4'],
        }

def createGraphWithData(data):
    g = Graph()
    for srcLabel, entries in data.items():
        print(srcLabel, entries)
        srcNode = addOrGetNode(g, srcLabel)
        for dstLabel in entries:
            dstNode = addOrGetNode(g, dstLabel)
            edge = g.addEdge()
            edge.connect(srcNode, dstNode)
```




```
def addOrGetNode(g, label):
    assert isinstance(g, Graph)
    if g.hasNode(label):
        return g.getNode(label)
    else:
        newNode = g.addNode(label)
        return newNode
```

Visualizing graph

- Use graphviz utility to plot graph into visual able format
- Use dot language to represent connectivity of vertices in graph
- Command line: -

```
dot -Tpng mygraph.dot -o mygraph.png
```

- The dot language can accept further attributes to tune its appearances. Visit gallery for examples
- There are few more rendering algorithms. The dot is one of such

```
digraph G {
1 -> 2
1 -> 5
3 -> 2
3 -> 4
2 -> 1
2 -> 3
2 -> 5
5 -> 1
5 -> 2
5 -> 4
4 -> 3
4 -> 5
4 -> 6
6 -> 4
}
```

Visiting graph the OO way

- Consider a CAD application that want to save its data to multiple formats (*.dwg, *.svg etc.).
  - Implementing each data save format in data model class can pollute the implementation of data model.
  - Visitor implementation (*.dwg, *.svg) may contain its own idiosyncrasy this further complicate the matter.
- Functions that simply examine a graph for nodes/ edges can be decoupled from the data structure hosting functions.
- Use visitor pattern to separate visiting implementations to a single visitor class.

Implementing visitor

```
class GNode:
    def accept(self, visitor):
        assert isinstance(visitor, Visitor)

        visitor.markSeen(self)
        for e in self._outEdges:
            e.accept(visitor)

        visitor.visit(self)
```

```
class GEdge:
    def accept(self, visitor):
        visitor.visit(self)
        if not visitor.hasSeen(self._outNode):
            self._outNode.accept(visitor)
```

```
class Graph:
    def accept(self, visitor):
        for n in self.getNodes():
            if not visitor.hasSeen(n):
                n.accept(visitor)
```

Implementing visitor (2)

```
class Visitor:
    def __init__(self):
        self._seen = set()

    def hasSeen(self, obj):
        return obj in self._seen

    def markSeen(self, obj):
        self._seen.add(obj)

    def visitEdge(self, obj):
        pass

    def visitNode(self, obj):
        pass

    def visit(self, obj):
        print("visiting obj", obj)
        if isinstance(obj, GEdge):
            self.visitEdge(obj)
        elif isinstance(obj, GNode):
            self.visitNode(obj)
```

```
class GraphVizWriter(Visitor):
    def __init__(self, outfile):
        super().__init__()
        self._out = open(outfile, 'wt')
        self._out.write("""digraph G {
concentrate=true
""")

    def __del__(self):
        self._out.write("}\n")

    def visitEdge(self, obj):
        assert isinstance(obj, GEdge)
        self._out.write("%s -> %s\n"
          % (obj._inNode, obj._outNode))

    def visitNode(self, obj):
        pass
```

```
def writeGraphVizFile(outfile, g):
    assert isinstance(g, Graph)
    writer = GraphVizWriter(outfile)
    g.accept(writer)
```

Notes about graph

- There are several ways to represent graph each with pros and cons usually for size versus performance tradeoff. The idea shown here belongs to adjacency list. Be creative!
- Graph generated randomly can be useful to generate graph inexpensively for testing. Of course there are many ideas to generate random graph. One idea is to connect edges at random so that there is probability *p* between two nodes. See Erdős–Rényi model

Exercises for graph

1. Complete the graph module by adding methods as suggested in here.
2. Create a visitor class to support writing Python data.
3. Extend the basic graph classes to support rich attributed graph.
4. Develop a parser that can parse graphviz data into graph. The parser may support additional attributes for visualization.
5. Develop a random graph generation routine that is based on the idea that any two nodes are connected by the probability of p. Hint: use random.random().
6. Develop an algorithm that will find the shortest path between two nodes. Use #3 to hilite the shortest path using graphviz.

Static methods

- Static methods are ordinary methods that just happen to live inside class namespace
- Defined using @staticmethod decorator

```
class Foo:
    @staticmethod
    def me():
        pass
```

Class methods

- Class methods behave just like static methods but take a first class argument
- Defined using @classmethod decorator
- In inheritance chain, the *cls* argument carries the the caller class

```
class A:
    @classmethod
    def me(cls):
        # cls carries A or B
…

class B(A): pass

class C(A): pass

B.me() # pass cls as B
C.me() # pass cls as C
```

Singleton pattern

- Occasionally it is useful to have a single instance throughout the lifetime of application
- Useful pattern for anything that should have a single occurrence e.g. factory, library, root window, manager
- The ideal singleton pattern also prevent initialization of singleton class
  - Python has no clear way to do this. Read more
  - Read more about singleton pattern in other language

```
class A:
    # class attributes
    _singleton = None

    @classmethod
    def sharedInstance(cls):
        if cls._singleton is None:
            cls._singleton = cls()
        return cls._singleton


class B(A): pass


class C(A): pass


# singleton of B
B.sharedInstance()

# singleton of A
C.sharedInstance()
```

Other ideas for Singleton and Class methods

- Use it for functions that do not change the state of the object
- Use it to do logical grouping of functions just like module do to cluster codes
- When class info is not needed prefer static method to class method
- Consider the choice against a top level function in module space well. Ask which one is clearer?

Object reference and copies

- Python (as with most scripting languages) assign by reference. On variable assignment only the **reference to the object is copied**!
- Garbage collection: object memory recycles when nil reference
- Advantage: Understandable copying object with a lot of data can become performance issue
- Watch out for side effects it can bite you!

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

def foo(c):
# c becomes reference to o
# in caller stack
    c.x = 10

o = Position(4,3)
# o = 4,3
foo(o)
# o becomes 10,3
# is this intentional??
```

Identity crisis: is and equality

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> a is b
False
>>> id(a)
4496831336
>>> id(b)
4496719024
```

```
>>> class Position:
...     …
...     def __eq__(self, other):
...         return self.x == other.x and self.y == other.y
...
>>> Position(4,3) is Position(4,3)
False
>>> Position(4,3) == Position(4,3)
True
```

- Python operator ‘is’ to check if two variables are **identical** refer to the same object
- Two objects may **not be identical** but **equivalent** because they contains the same values

Exercises for Singleton

The copy module

```
import copy
…
o = Position(4,3)
# o = 4,3
foo(copy.copy(o))
# no side effect to o
```

- Shallow copy - copy creates a new object and copies all the source object references to the new object
  - The content in new object shares the same reference as with original source.
- Use copy.deepcopy() to do recursive copy
- If copying is critical, overload __copy__ and__deepcopy__ method in your class that encapsulate your custom copy recipes
  - Give your class a copy() method.

Object mutability

- Python's primitives data type, e.g. numbers and strings are immutable (non writable).
  - A new object has to be created for every change!
  - String is a immutable data container for a sequence of characters
  - Tuple behaves like list but differs on mutability
- Objects are usually mutable unless special care taken care to design for immutable

Immutable object

- Advantages: -
  1. Eliminate side-effects, easier to debug
  2. Easier for threading
  3. Favorite data type for functional programming
- Simpler code, can save hours of debugging!

```
from collections import namedtuple

class Point(namedtuple('Point', 'x y')):
    def __repr__(self):
        return "Point({}, {})".format(self.x, self.y)

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

def foo(c):
    assert isinstance(c, Point)
# error
# AttributeError: can't set attribute
    c.x = 3
```

Immutable object design ideas

- You usually want your lowest data type to be immutable: e.g. rectangles, polygons, vectors, complex numbers etc.
- Use tuple or namedtuple for attributes in your class
- For collection data, overrides the __setitem__ to block assignment e.g. raising TypeError
- Mutable class can be viewed as subset of immutable class with added mutable operations

Exercises for object mutability

1. Identify any class in your solution that could have been implemented as immutable object?
2. What are your reasoning behind your selection?

Magic methods

- Predefined methods that start and end with '__' have magical properties
  - ❖ __init__() is a magic method!
- Consider these: -
  1. Using '+' operator to join two list.
  2. Using '[]' operator to fetch an object with key or index from a container.
  3. len() function that will return size of the object.
- Main usages: -
  - Provides overloaded operation to Python's operator or built-in function
  - to define otherwise undefined behavior

```
>>> [1,2,3] + [4,5,6]
[1, 2, 3, 4, 5, 6]

>>> a = [1,2,3]
>>> a.extend([4,5,6])
>>> a
[1, 2, 3, 4, 5, 6]
```

Example using magic methods

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

    def __repr__(self):
        return "Position({}, {})".format(self.x, self.y)

    def __add__(self, other):
        return Position(self.x + other.x, self.y + other.y)

a = Position(3,2) + Position(1,1)
# a gets Position(4, 3)
```

```
>>> import numpy as np
>>> a = numpy.matrix('1 2; 3 4')
>>> a * 2
matrix([[2, 4],
        [6, 8]])
```

Practical use of magic methods

- Native syntax that works on your custom class. Natural syntax leads to more readable code.
  - Think Position(2,3) + Position(3,5), matrix operations
- Be able to subscribe to Python's programming framework simply by implementing the required magic method: -
  1. Define __lt__ such that object become sortable using sorted() function.
  2. Implement __iter__ and __next__ so that object can participate in Python iterable protocol.
  3. Implement __contains__ so that object behave like a Python's built-in container.
- Use collections.abc for common Python collection type.
- See also Rafe's article for excellent guide to Python's magic methods!

Exercises – Give some magic to your class

- Refer to Python's collections.abc doc, find out the closest collection kind that describes Graph class. Explain your rationales.
  - Complete the required magic methods implementation.
  - Test your solution.

```
import collections.abc

class Graph(collections.abc.XYZ):
  …
```

Writing functions

- Multi-paradigm programming languages
- Generic programming
- Functional programming

Generic programming

to describe a programming paradigm whereby fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalised as concepts, with generic functions implemented in terms of these concepts

In generic programming, a **concept** is a description of supported operations on a type, including syntax and semantics. In this way, concepts are related to abstract base classes but concepts do not require a subtype relationship.

Source Wikipedia

- In OOP, you define a set of operations for a class of an object
- In GP, for a given function you specify a set of requirements to one or several input objects

GP motivations

- Implementing a set of algorithms for each object type will require MxN implementations (assuming M = number of type, N = number of algorithms). For example rather than having a sort function for each data type (int, string, custom class), is it viable to implement once for every data type?
- Thus generic function can scale across multiple data type
- Large ROI in code reuse when algorithm is complex and ability to scale wide data range

Python generic functions

- Python itself has adopted GP widely, many trivial and non-trivial: -
  1. The len() function simply query the object for implemented __len__ method
  2. String-able object implements the __str__ method such that any object has a string presentation
  3. Function that takes iterable object is expecting object to implement iteration protocol via the built-in functions iter() and next(), more about this …
  4. sorted() expects all object in sequence to implement lesser than operator (<)

Sorted()

```
# better yet
#@functools.total_ordering
class MyObject:
    def __init__(self, w):
        self._w = w

    def __str__(self):
        return 'MyObject(%s)' % self._w

    def __lt__(self, other):
        assert isinstance(other, MyObject)
        return self._w < other._w
```

```
# sort objects in ascending order
for x in sorted(map(MyObject, [8, 3, 6, 9, 10, 1])):
    print(x)

print(max(map(MyObject, [8, 3, 6, 9, 23, 1])))
# MyObject(23)
```

- Sort() expects object to implement lesser than < operator
- Failure to comply will result in error: -
  - TypeError: unorderable types: MyObject() < MyObject()

Sample generic function

```
def qsort1(list):
    """Quicksort using list comprehensions"""
    if list == []:
        return []
    else:
        # list must be indexable
        pivot = list[0]

        # list must be iterable, in addition must support
        # < and >= operators
        lesser = qsort1([x for x in list[1:] if x < pivot])
        greater = qsort1([x for x in list[1:] if x >= pivot])
        return lesser + [pivot] + greater
```

GP the concept

- Like in business, a biz represents an agreeable contract between client and supplier of goods and services
- In GP a class can have varying implementation. So long it has consistent interface and behavior the designated generic functions will work!
  - Like in biz, implementations or executions are considered internal matters.
  - In GP framework, generic function asserts requirements automatically to input arguments to deliver its promised functionality
- Like lawyers ☺ in GP most effort were spent on formalizing such requirements
- Other similar concepts: mixin class, aspect oriented programming.

GP challenges in Python

- Python is a typeless language – with “duck typing” a function will behave if object implements all the required methods
- It can become problematic to track all required methods involved in a contract in you brain!

☺ They are helpers to make GP easier. The abc metaclass
☺ Improved readability of code considers *foo(mappable)* rather than *foo(x)*
☹ No compile static checking
☺ Things are looking better in py3.4 (PEP 443)

How to use ABC

```
import abc

class HNode(metaclass=abc.ABCMeta):

    def getChildren(self):
        raise NotImplementedError
    children = abc.abstractproperty(getChildren)

    @abc.abstractmethod
    def appendChild(self):
        raise NotImplementedError

    def getParent(self):
        raise NotImplementedError
    def setParent(self, parent):
        raise NotImplementedError
    parent = abc.abstractproperty(getParent, setParent)
```

```
# … helper method in HNode too
    def sumOfSize(self):
        total = 0
        for c in self.children:
            assert isinstance(c, HNode)
            total += c.sumOfSize()
        return total

class SomeBaseNode:
    …

class SomeNode(SomeBaseNode, HNode):
    def __init__(self):
        self._parent = None

    def getParent(self):
        return self._parent
    def setParent(self, parent):
        self._parent = parent

    parent = property(getParent, setParent)
```

How to use ABC (2)

```
# attempt to construct an object from incomplete class definition
a = SomeNode()
TypeError: Can't instantiate abstract class SomeNode with abstract methods children, parent, size

# generic function to return children of a node
def children(node):
    assert isinstance(node, HNode)
    return node.children

# generic function to write graphviz dot file??
def writeDotFile(root, file):
    …
```

- Automated type checking
- IDE auto-complete assistance
- Understandable improving readability

ABC for common collections

- Use collections.abc to simplify building common containers.
- Library of abc for common container e.g. container, hashable, iterable, iterator etc.
- The Set ABC auto provides the rest of Set requirements given __contains__, __iter__ and __len__.
- Use this to build higher ordered collection based class.

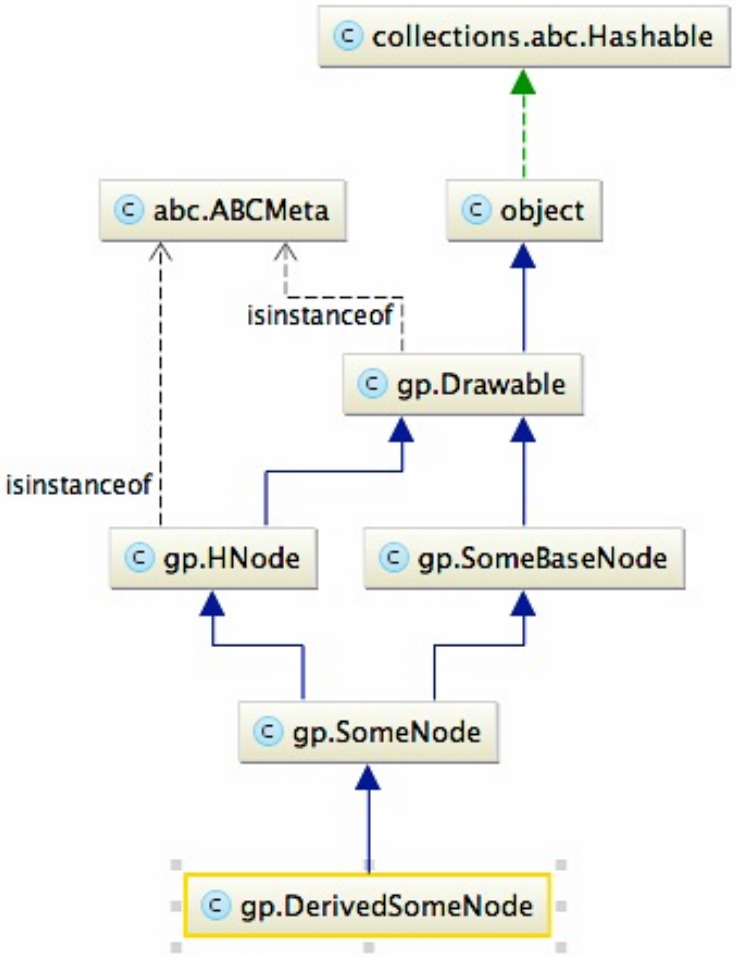
Thinking in GP

- Identify functions that you have to repeat for every data structure type. Replicating functions for different data structure are the best candidates for GP
- Think about object at abstract level for repeating pattern. For example both rect and circle are *drawable*
- More tips: -
  - Reoccurring concept – Try to put a verb to the problem *-able e.g. Hashable, Drawable, Sortable etc.
  - GP can often be confused with OOP abstract technique. Hint try to cross-cut concerns to break it to smaller problem set.
    - Plants and animals are both living being but plants are largely stationary. So you may have Stationary and Mobility as ABC.
    - A class participating in GP can inherit many characteristics.
  - Protocol – Collaborate via a sequence of message passing between two parties just like network protocol. E.g. iteration protocol in Python

Recalling super()

```
class Drawable(metaclass=abc.ABCMeta):
    def draw(self):
        print("draw in Drawable")

class HNode(Drawable, metaclass=abc.ABCMeta):
    def draw(self):
        super().draw()
        print("draw in HNode")

class SomeBaseNode(Drawable):
    def draw(self):
        super().draw()
        print("draw in SomeBaseNode")

class SomeNode(SomeBaseNode, HNode):
    def draw(self):
        super().draw()
        print("draw in SomeNode")

class DerivedSomeNode(SomeNode):
    def draw(self):
        super().draw()
        print("draw in DerivedSomeNode")
```

Which order you would like to go first?



Recalling super() (2)

```
>>> print("DerivedSomeNode mro is:", DerivedSomeNode.__mro__)
DerivedSomeNode mro is: (<class '__main__.DerivedSomeNode'>,
<class '__main__.SomeNode'>,
<class '__main__.SomeBaseNode'>,
<class '__main__.HNode'>,
<class '__main__.Drawable'>,
<class 'object'>)
```

```
>>> d = DerivedSomeNode()
>>> d.draw()
draw in Drawable
draw in HNode
draw in SomeBaseNode
draw in SomeNode
draw in DerivedSomeNode
```

- MRO – method resolution order
- Python search this list in order for method to dispatch (it applies to attributes too!)
- Python has 3 implementation before resolved to one - C3 linearization
- C3 linearization conceptually: -
  1. Go as deep as possible first - from the most specialized to most generic.
  2. When there is a common class (e.g. Drawable class) fall back to the most specialized node, visit the next node in order, repeat step #1

Multi-inheritance good practices

1. Pre-process super() first (simply place super in front).
   - ◆You are likely want to process the base first before more specialized methods come.
   - ◆Tune the order later
2. Make sure the super method exists in all participating class
3. Make sure there is matching argument for the super method. Use *args, *kwargs as your friend
4. Remember to call super to continue the processing chain. Delete it to stop the processing chain

Exercises for GP

1. Complete the HNode implementation using ABC class.
2. Apply HNode abstract class to graph and tree based on previous exercise.
3. Implement a generic function to write graphviz dot file. Use random graph or tree generation routinne developed in previous exercise to test your solution.

Metaclasses

Exceptions

```
def foo():
    …
    # error condition
    return -1
```

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

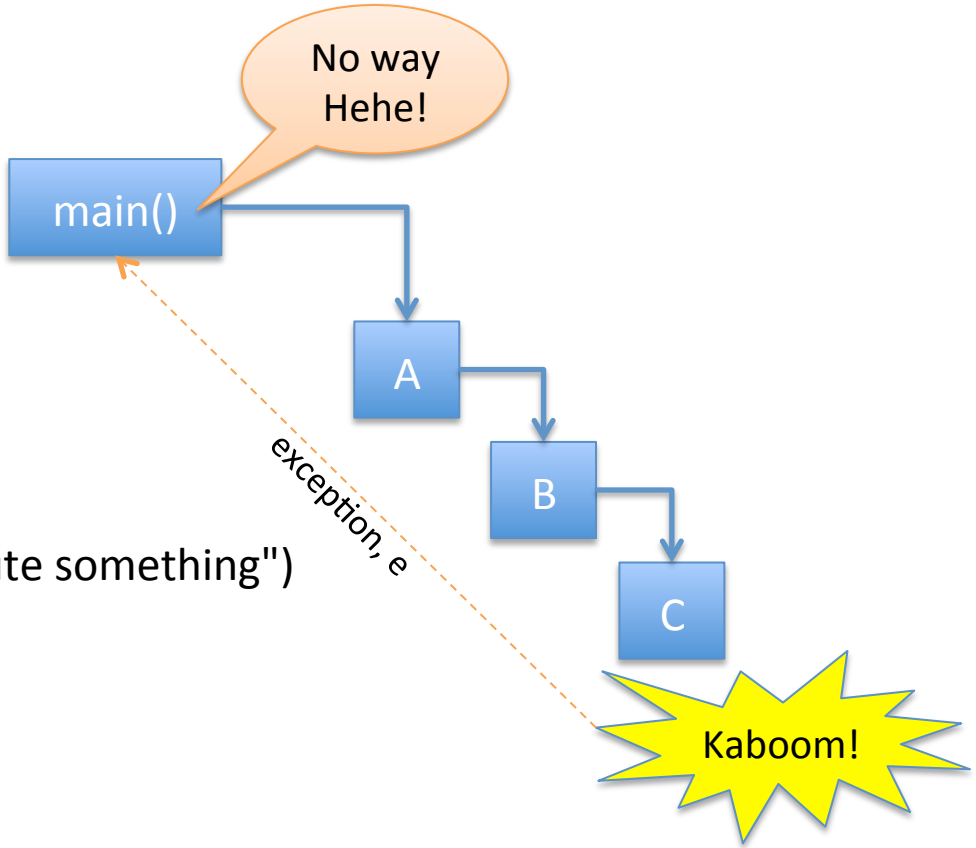
- Traditional error handling via return value
- Exception changes the course of normal execution flow
- Program can be designed to response to exception with alternate program flow or logic typically to recover from error condition

Python and exceptions

- Python itself has been practicing exception throughout: -
  - SyntaxError raised when source code has invalid syntax
  - KeyError raised when attempt to access dictionary with non-existent key. To test a key properly use: *x in container*
  - For complete list
- App may choose to raise built-ins exception without reinventing new one (for clarity)

Handling exception

```
def A():
    return B()

def B():
    return C()

def C():
    # error 1
    raise KeyError('key error')
    # error 2
    raise OverflowError("can't compute something")
try:
    A()
except OverflowError as e:
    # handle overflow error here
# empty or more exception …
except: # leave out the argument for “catch-all”
   # handle every else here
finally:
   # clean-up code
```



Unhandled exception

```
Traceback (most recent call last):
  File "/Users/bennykhoo/Sources/MyPythonClass/except.py", line 15, in <module>
    A()
  File "/Users/bennykhoo/Sources/MyPythonClass/except.py", line 4, in A
    return B()
  File "/Users/bennykhoo/Sources/MyPythonClass/except.py", line 7, in B
    return C()
  File "/Users/bennykhoo/Sources/MyPythonClass/except.py", line 10, in C
    raise KeyError('key error')
KeyError: 'key error'
```

- You will get stack trace which can be useful for debug
- What is your opinion?

Exception is just ordinary object

- Exception is just a regular of class with a constructor
- Always derive from base Exception class or below to create custom exception class just like ordinary OOP

- Good use cases if you want to carry other error info as attributes
- Keep it simple

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print('My exception occurred, value:', e.value)
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Finally …

- The finally block in try statement
- Must be executed under all circumstances whether exception has occurred or not
- Good use for handling clean-up
  - Closing hardware handles which occupied system resources
  - Critical exit point such transaction exit in security application
  - Releasing locks in multi threaded code

The with statement

```
# old style
f = open('debuglog', 'a')
f.write("debugging\n")
statements
exception occur!!
f.close()
```

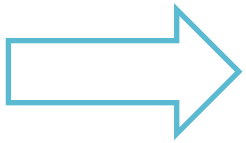
```
with open('debuglog', 'a') as f:
    f.write("debugging\n")
    statements
    f.write("done\n")
# file f is guaranteed closed
```

```
import threading
lock = threading.lock()
with lock:
    # critical section
    statements
    # end critical section
# lock is guaranteed released
```

- Managing system resources can be tricky biz
  - Error condition can cause control flow to bypass statement responsible for proper releasing of resources
- A form of defensive programming pattern [link]

With and without

```
with VAR = EXPR:
    BLOCK
```



roughly equivalents to

```
VAR = EXPR
VAR.__enter__()
try:
    BLOCK
finally:
    VAR.__exit__()
```

- A short hand idiom for try-except-finally block
- Magic methods __enter__ and __exit__ for context management protocol

Custom context management

```
class KiasuListTransaction:
    def __init__(self, mylist):
        self.mylist = mylist

    def __enter__(self):
        self.workingcopy = list(self.mylist)
        return self.workingcopy

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is None: # yeah!
            self.mylist[:] = self.workingcopy

        # False if any exception occur during exit is propagated out
        return False
```

- If no exception has been raised during the with block - the control flow passes None for the three arguments exc_type, exc_val, exc_tb

Custom context management (2)

```
a = [1,2,3]
with KiasuListTransaction(a) as working:
    working.append(4)
    working.append(5)

# should print [1, 2, 3, 4, 5]
print(a)
```

```
a = [1,2,3]
try:
    with KiasuListTransaction(a) as working:
        working.append(4)
        working.append(5)
        raise RuntimeError('uh oh!')
except RuntimeError as e:
    print(e, "... alrite we are backing up")

# should print [1, 2, 3]
print(a)
```

- When exception has occurred inside the with block, changes are not committed back to original list

Context manager

```
from contextlib import contextmanager

@contextmanager
def KiasuListTransaction(mylist):
    workingcopy = list(mylist)
    yield workingcopy
    # if exception has occurred inside with-block execution will not pass this line
    mylist[:] = workingcopy
```

- Easier way to write context manager

Case study

First class everything

> I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, etc.) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth.
> ~ gvr

- Everything that can be named in Python are “first class objects”
- A first class object is an entity that can be dynamically created, destroyed, passed to a function, returned as a value, and have all the rights as other variables in the programming language have. [link]
- In a nutshell: all nameable constructs in Python can be treated like an ordinary object

what does first class means?

```
class Foo:
    def instanceMethod(self):
        pass

    @classmethod
    def staticMethod(cls):
        pass

def func():
    pass
```

```
(Pdb) func # ordinary function
<function func at 0x10bf0c7a0>

(Pdb) Foo # ordinary class
<class '__main__.Foo'>

(Pdb) Foo.instanceMethod # ordinary function
<function Foo.instanceMethod at 0x10bedee60>

(Pdb) Foo.staticMethod
# a bounded function with the first argument fixed to Foo
<bound method type.staticMethod of <class '__main__.Foo'>>

(Pdb) x = Foo()
(Pdb) x.instanceMethod
# is a bounded function with the first argument fixed to inst x
<bound method Foo.instanceMethod of
<__main__.Foo object at 0x10bf2bcd0>>
```

Literals are first class too!

```
>>> dir(123)
['__abs__', '__add__', '__and__', ’ …

>>> dir("abc")
['__add__', '__class__', '__contains__ …

>>> "1,2,3".split(',')
['1', '2', '3’]

>>> type("1,2,3")
<class 'str’>
```

- Use dir() function to list all attributes
- That explains why the dot ‘.’ operator works on literals

Function as argument

```
def fib(n):
    if n is 0 or n is 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def trace(f):
    f.indent = 0
    def g(x):
        print('|  ' * f.indent + '|--', f.__name__, x)
        f.indent += 1
        value = f(x)
        print('|  ' * f.indent + '|--', 'return', repr(value))
        f.indent -= 1
        return value
    # register name of f to interpreter
    # setattr(sys.modules[__name__], f.__name__, g)
    return g
fib = trace(fib)
print(fib(4))
```

```
$ python fib.py
|-- fib 4
|  |-- fib 3
|  |  |-- fib 2
|  |  |  |-- fib 1
|  |  |  |  |-- return 1
|  |  |  |-- fib 0
|  |  |  |  |-- return 1
|  |  |  |-- return 2
|  |  |-- fib 1
|  |  |  |-- return 1
|  |  |-- return 3
|  |-- fib 2
|  |  |-- fib 1
|  |  |  |-- return 1
|  |  |-- fib 0
|  |  |  |-- return 1
|  |  |-- return 2
|  |-- return 5
5
```

Python decorator

```
@trace
def fib(n):
    if n is 0 or n is 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

print(fib(4))
```

- Python decoration is a function that takes a function to return a modified function
- The '@' apply a decorating function to the following function

How about decorating a class?

- Class decorator is a function that takes a class as input and return a modified class
- How is it possible? Class is a first class object which means you can process it just like any ordinary object: -
  - Use getattr(cls, name) and setattr(cls, name) to fetch attribute in class, define new method or modify existing method in a class.
  - For example the functools.total_ordering is a built-in class decorator which takes a class with any one of the < > <= >= method to complement for the rest of missing operator method
- Can be powerful but need to exercise care, can lead to excessive magical class

Class decorator example

```
def trace(class_):
    original_init = class_.__init__
    def __init__(self, *args, **kws):
        print("Instantiating an object of class {}".format(class_.__name__))
        original_init(self, *args, **kws)
    class_.__init__ = __init__
    return class_

@trace
class Foo(object):
    def __init__(self, value):
        self.value = value

foo = Foo(5)
print("The value of foo is {}".format(foo.value))
```

Iterator pattern

- Given a collection of any data kinds: -
  - How do I access individual component in the collection?
  - If order is necessary, what is the desirable order in the access sequence?
- Iterator pattern helps to decouple data traversing algorithms from the implementation in data host.
- Applications of iterator pattern: -
  - In data structure, what is the next node in a bfs or dfs order in a given graph?
  - In geometry, what are all possible free spaces in a given plot? What are the objects closest to the farthest from a given object?
  - In scheduling, what is my next most important task in my schedule?
  - In maths, what is the next value in an infinite series? Think fibonacci, fourier or prime number series

Understanding iteration in Python

- *iter()* function to create an iterator object
- *next()* function to return the next element in the sequence while incrementing the state of iterator object to the next state
- When an iteration is exhausted, the iterator object raise a *StopIteraton* exception
- The *for* loop statement does the following internally: -
  1. Invokes *iter(obj)* to create an iterator object
  2. On iteration, invokes *next(iter)* to iterate through the sequence
  3. Terminates the iteration when the *iter* object raises StopIteration

```
>>> L = [1,2,3]
>>> it = iter(L) # same as L.__iter__()
>>> it
<list_iterator object at 0x104dab190>
>>> it.__next__()  # same as next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```

Clarifying iterable, iterator and iteration in Python

- An iterator object contains current state of iteration and implements the __next__() method to increment iteration to the next state
- An object that can produce an iterator object is called iterable object. A typical iterable object implements the __iter__() method to return an iterator object
- In *for* statement (*for X in Y*), Python expects the Y as an iterable object: -
- A typical iterator object returns itself when attempting to ask for iterator object. Hence an iterator object can be seen having **is-a** relationship to iterable class.
  - Because of this reason, most algorithm functions choose to take an iterable object rather than iterator object

Implementing iterator the OO way

- Provides implementation for the two magic methods __iter__ and __next__
- Unless it is an infinite sequence make sure to raise StopIteration exception when the sequence is exhausted
- Another possible implementation: -
  1. Implement __iter__ in the collection object (e.g. graph) to produce an iterator object. Think dfs_iterator and bfs_iterator
  2. Implement __next__ in iterator object
  3. __iter__ implementation in an iterator object can simply return itself for good practice.

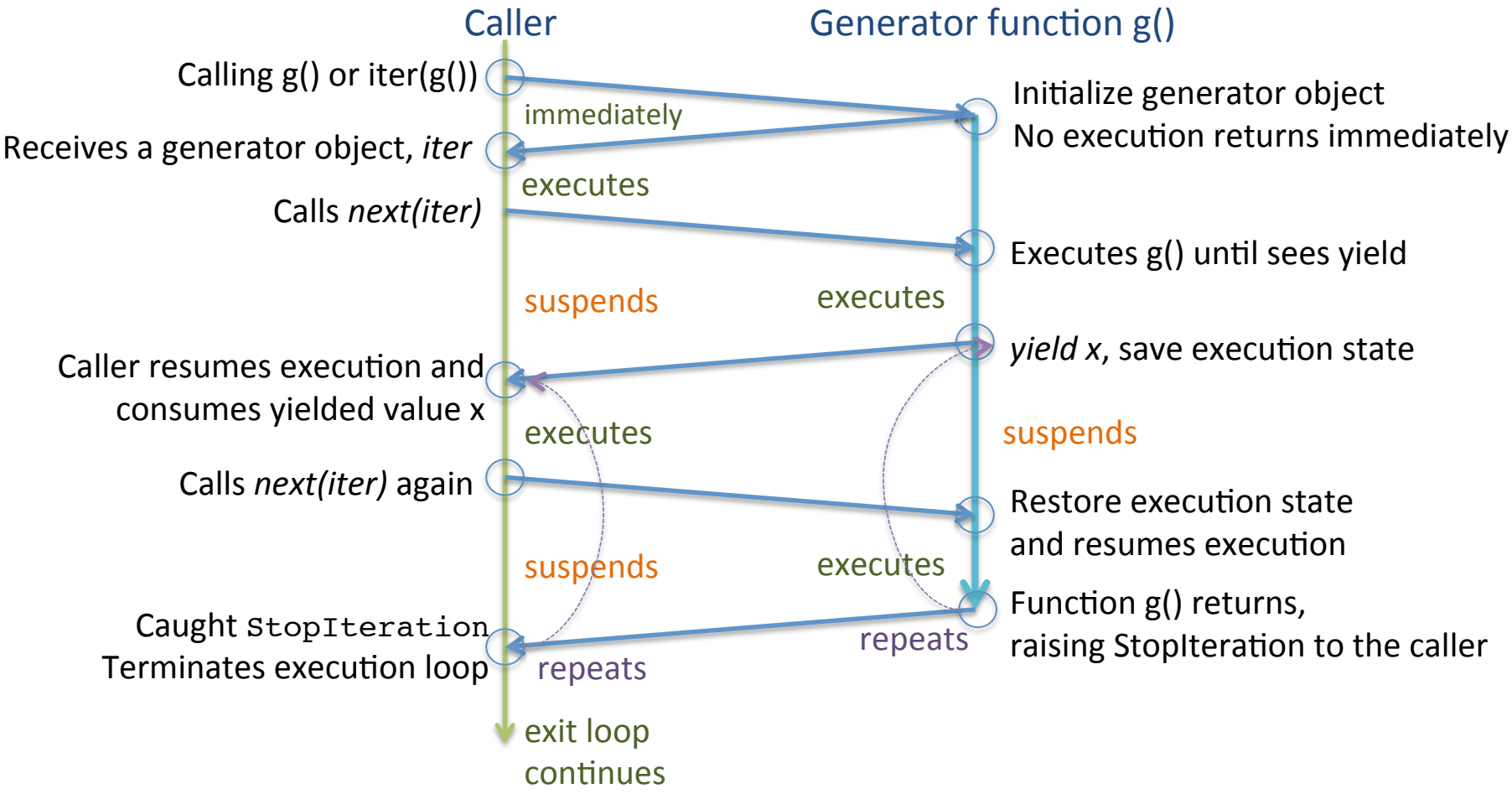
```
Class CountingIterator:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

for c in CountingIterator (3, 8):
    print(c)
```

Simplifying iteration with Generator

```
>>> def generate_ints(N):
…    print(“Init generator”)
...    for i in range(N):
...        yield i
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> next(gen)
Init generator
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most recent call last):
  File "stdin", line 1, in ?
  File "stdin", line 2, in generate_ints
StopIteration
```

- Special class functions that simplify the tasks for writing iterators
- Regular functions compute a value and return it.
- Generator functions return an iterator object that returns a sequence of values

Generators collaboration diagram



Infinite series

- Many mathematical series can be formulated as a generator function with indeterminate limit
- Do you see the motivation?

```
def generateFibNum(limit=100):
    n2 = 1
    n1 = 2
    yield n2
    yield n1
    while True:
        x = n1 + n2
        if x > limit:
            break
        yield x
        n2 = n1
        n1 = x
```

Revisiting tree traversal

```
def elementsInTree(n):
    """yield all nodes in a given tree node"""
    assert isinstance(n, TreeNode)

    yield n
    for childNode in n.getChildren():
        # recurse in childNode
        # < py3.3, must yield again
        # for childNode2 in elementsInTree(childNode):
        #     yield childNode2
        # >= py3.3 only
        yield from elementsInTree(childNode)

# test iterator
for n in elementsInTree(root):
        # process n
        pass
```

```
def leavesInTree(n):
    """yield all leaves in a tree"""”

    # if node has no children it is a leaf
    if len(n.getChildren()) == 0:
        yield n

    for childNode in n.getChildren():
        # recurse in childNode
        yield from leavesInTree(childNode)
```

Revisiting graph traversal

- Two popular algorithms for traversing graph:
  - DFS (Depth-first search)
  - BFS (Breadth-first search)

Exercises for generator

- Write a generator function to iterate the graph in DFS and BFS order one for each.

List comprehension

- Without list comprehension

```
nums = [1, 2, 3, 4, 5]
squares = []
for n in nums:
    squares.append(n * n)
```

- Using list comprehension

```
>>> squares = [n*n for n in [1, 2, 3, 4, 5]]
>>> squares
[1, 4, 9, 16, 25]
```

- Build a list like mathematical set theory
- {x | x > 10} refers to the set for all x greater than 10
- List comprehension is faster than lambda!

List comprehension with condition

- You may attach ifs to list comprehension too.

```
stripped_list = [line.strip() for line in line_list if line != ""]
```

```
( expression for expr in sequence1
        if condition1
        for expr2 in sequence2
        if condition2
        for expr3 in sequence3 ...
        if condition3
        for exprN in sequenceN
        if conditionN )
```

```
for expr1 in sequence1:
    if not (condition1):
        continue   # Skip this element
    for expr2 in sequence2:
        if not (condition2):
            continue    # Skip this element
        ...
        for exprN in sequenceN:
            if not (conditionN):
                continue   # Skip this element

            # Output the value of
            # the expression.
```

Quicksort using list comprehension

```
def qsort1(list):
    """Quicksort using list comprehensions"""
    if list == []:
        return []
    else:
        pivot = list[0]
        lesser = qsort1([x for x in list[1:] if x < pivot])
        greater = qsort1([x for x in list[1:] if x >= pivot])
        return lesser + [pivot] + greater
```

Generator Expression

```
>>> g = (n*n for n in someInfiniteSequence())
>>> g
<generator object <genexpr> at 0x10b0958c0>
>>> next(g)
1
>>> next(g)
4
```

- Use bracket () instead of []
- Works like list comprehension but creates an iterator instead
- Evaluate the next value **lazily**. Can be useful for infinite sequence which is impossible to build using list comprehension

Generator Expression motivation

- Useful to express a sequence of values that is infinite e.g. prime numbers.
- Faster. List comprehension have to build the list first prior to pass to f() as a list argument.
- f() evaluates the generator expression immediately. Less memory used

```
# using list comprehension
sum([x**2 for x in range(1, 11)])

# or generator expression
sum(x**2 for x in range(1, 11))
```

Functional programming

Currying

Partial function

Function overloading

Higher order functions

Multi-method dispatching

First-class functions

Lambda

Generic functions

Functional composition

Tail recursion

Declarative programming

Mutable data

Lazy evaluation

Pure functional model

$$y = f(x)$$

- All variables are not mutable at best!
  - Bounded variables are not writable
  - Only unbounded variables are assignable
- No loops - use function or recursion instead
- Conditional `ifs' becomes function
  - alternatively use multifunction dispatching

FP advantages

- Biggest wins in concurrency [ref] – data race
  - Therac-25 disaster radioactive overdoses
  - Northeast blackout 2003
- Brevity
- Was a programming language for academia but it is slowly gaining acceptance
  - In telecom industry or Ericsson only?
  - Do you know Whatsapp?
- Functional features were ported gradually into imperative languages since 90s.
- But Python is a multi-paradigm languages
- Learn you some pure FP languages: Erlang and Haskell

A programming task

Calculate partially invalid string with operations:
"28+32+++32++39”

* Sample was adopted based on excellent presentation about Python FP

Imperative style

Actions that change state from initial state to arrive at result

```
expr, res = "28+32+++32++39", 0
for t in expr.split("+"):
    if t != "":
        res += int(t)

print res
```

```
"28+32+++32++39", 0
"28", 0
"32", 28
"", 60
"", 60
"32", 60
"", 92
"39", 92
131
```

Functional style

```
from operator import add
expr = "28+32+++32++39"
print reduce(add, map(int, filter(bool, expr.split("+"))))
```

```
"28+32+++32++39"
["28","32","","","32","","39"]
["28","32","32","39"]
[28,32,32,39]
131
```

Languages for FP: map()

map(f, iterA, iterB, ...) returns an iterator over the sequence f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ...

```
>>> def upper(s):
...     return s.upper()

>>> list(map(upper, ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
>>> [upper(s) for s in ['sentence', 'fragment']]
['SENTENCE', 'FRAGMENT']
```

- Transform input data stream by *f*

Languages for FP: filter()

filter(predicate, iter) returns an iterator over all the sequence elements that predicate will evaluate items in iterator to truth

```
>>> def is_even(x):
...     return (x % 2) == 0

>>> list(filter(is_even, range(10)))
[0, 2, 4, 6, 8]

# list comprehension
>>> list(x for x in range(10) if is_even(x))
[0, 2, 4, 6, 8]
```

- Able to select data that qualify certain criteria without touching internal implementation of a host function

Languages for FP: any() and all()

any() is like the BIG version of OR logic
all() is like the BIG version of AND logic

```
>>> any([0,1,0])
True
>>> any([0,0,0])
False
>>> any([1,1,1])
True
>>> all([0,1,0])
False
>>> all([0,0,0])
False
>>> all([1,1,1])
True
```

any() and all(): usages

any() – Test if a sequence contains something

```
>>> def is_float(x):
...     return isinstance(x, float)
...
>>> any(map(is_float, [3, 4.5, 2, 3.14]))
True
>>> any(map(is_float, [1, 2, 3, 4]))
False
```

Languages for FP: Small functions are good functions - lambda

You'll often need little functions that act as predicates or that combine elements in some way.

```
>>> list(map(lambda s: s.upper(), ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
```

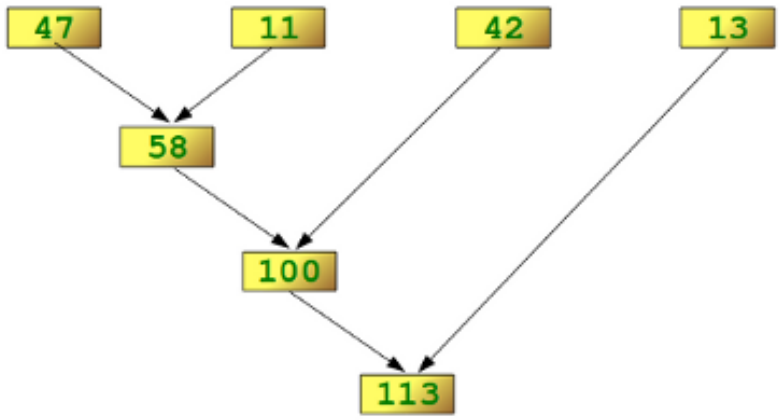
- Lambda is worth only one line of code. If larger function is needed fall back to conventional function
- You may pass lambda along to map, filter, sorted or any functions that accept a function as input

Languages for FP: functools.reduce()

Apply function of two arguments cumulatively to the items of sequence, from left to right, so as to reduce the sequence to a single value

reduce(f, [a, b, c, d]) => f(f(f(a,b),c),d)

```
>>> from functools import reduce
>>> reduce(lambda x,y: x+y, [47,11,42,13])
113
>>> sum([47,11,42,13])
113

# find common list
>>> input_list = [set([1,2,3,4,5]),
                  set([2,3,4,5,6]),
                  set([3,4,5,6,7])]
>>> reduce(set.intersection, input_list)
{3, 4, 5}
```



Languages for FP: few more

- You've just covered most commonly used function used in FP
- There are few more remaining in functools module that could be useful to you
- Checkout itertools for functions that process iterable in interesting ways

FP: closure

- A closure occurs when a function has access to a local variable from an enclosing scope that has finished its execution.
- A closure function is created with the local variables resolved (frozen) at the context where its was executed in runtime.

```
def add(a, b):
    return a + b

def my_partial(f, *args):
    def _(*a):
        new_args = args + a
        return f(*new_args)
    return _

# bind 5 to my add function
add5 = my_partial(add, 5)
add5(3)
# 8
```

- Gives us the ability to defer computation
- Important concept in FP to build higher ordered function.

Languages for FP: functools.partial()

The constructor for partial() takes the arguments (function, arg1, arg2, ..., kwarg1=value1, kwarg2=value2). The resulting object is callable, so you can just call it to invoke function with the filled-in arguments.

```
def isnumber(value):
    return any(map(partial(isinstance, value), [float, int]))
```

- Vary existing function by fusing its arguments
- Useful to produce variety without touching internal implementation of existing function

FP: currying

… is the technique of transforming a function that takes multiple arguments in such a way that it can be called as a chain of functions, each with a single argument [ref]

$$f(x, y) = y / x$$

The curry version of *f* becomes: -

```
def h(x):
    def g(y):
        return y / x
    return h
```

```
>>> h(2)(1)
0.5
>>> h(2)
<function g.<locals>.h at 0x02A6D618>
```

FP: currying and multiple arguments

```
def f(a, b, c, d, e):
    return a + b + c + d + e
```

```
def f(a):
    def g(b):
        def h(c):
            def i(d):
                def j(e):
                    return a + b + c + d + e
                return j
            return i
        return h
    return g

f(1)(2)(3)(4)(5)
```

FP: function that generates function

- Currying belongs to the class of function that generates function
- Function that generates function is common in natural world: -
  - ❖DNA is a function that replicates itself to generate proteins
  - ❖Many maths theories are concerning transformation of function forms e.g. calculus
  - ❖Cellular automata
- For us we are interested on the thinking pattern!

Case study: A programming task

Develop a calculator program that will parse a generic equation in string eventually solving it.

a. If the equation is solvable then the program should compute its equivalent.
b. If the equation contains variables generate a function instead that will solve the equation with the variables as arguments

"1 + 2*3 + 4" => 11

"2*x + y" => f
f(x=3, y=1) => 7
f(x=2, y=3) => 7

"a^2 + b^2" => g
g(a=3, b=4) => 25

How compiler solves an expression?

- Are you able to identify the pattern?

"1 + 2*3 + 4"  (1 +)(((2*)(3) +)(4))

- Eventually there are only one to two arguments to deal with at any instance of time

Partial evaluation

When a term cannot be solve, defer it to a function

((2*)(x) +)(y)

2*x => f(x)

f(x)+y => g(x, y)

a^2 + b^2

a^2 => f(a)

b^2 => g(b)

f(a) + g(b) => h(a, b)

Romance of Three Kingdoms

F * F -> G(*)
F * *x* -> G(*x*, *)
F * *d* -> G(*)

x * F -> G(*x*, *)
x * *y* -> G(*x*, y, *)
x * *d* -> G(*x*, *)

d * *d* -> *d*
d * *x* -> G(*x*)
d * F -> G(*)

- Symbol or Variable?
- Function or Functor?
- Number or Terminal or Literal?

Implementing Functor

```
from types import FunctionType

class Functor:

 def __init__(self, value):
       assert isinstance(value, FunctionType)
       self._value = value

# so that f() is workable
 def __call__(self, *args, **kwargs):
       return self._value(**kwargs)
```

```
...
 def __mul__(self, other):
       # f * t
       if isterminal(other):
          def _(**kwargs):
             return self(**kwargs) * other
          return Functor(_)
       # f * x
       if isinstance(other, Variable):
          def _(**kwargs):
             a = kwargs[other._name]
             return self(**kwargs) * a
          return Functor(_)
       # f * f
       if isinstance(other, Functor):
          def _(**kwargs):
             return self(**kwargs) * other(**kwargs)
          return Functor(_)
       assert False, "{} invalid other {}".format(self, other)
```

Where are the rest of the operators?

```
import operator

class BasicOp:
    def __add__(self, other):
        return self.apply(operator.add, other)
    def __sub__(self, other):
        return self.apply(operator.sub, other)
    def __mul__(self, other):
        return self.apply(operator.mul, other)
    def __truediv__(self, other):
        return self.apply(operator.truediv, other)
    def __pow__(self, power, modulo=None):
        return self.apply(operator.pow, power)

class Functor(BasicOp):
    def __mul__(self, other):
    def apply(self, op, other):
...
```

Implementing Variable

```
class Variable(BasicOp):
    def __init__(self, name):
        self._name = name

    def apply(self, op, other):
        # x * t
        if isterminal(other):
            def _(**kwargs):
                a = kwargs[self._name]
                return op(a, other)
            return Functor(_)
        # x * x
        if isinstance(other, Symbol):
            def _(**kwargs):
                a = kwargs[self._name]
                b = kwargs[other._name]
                return op(a, b)
            return Functor(_)
```

...

```
        # x * f
        if isinstance(other, Functor):
            def _(**kwargs):
                a = kwargs[self._name]
                return op(a, other(**kwargs))
            return Functor(_)

        assert False, "{} invalid other {}".format(self, other)
```

Implementing Number

```
class Number(Terminal, BasicOp):
    def __init__(self, value):
        assert isnumber(value)
        self._value = value

    def apply(self, op, other):
        # n * d
        if isnumber(other):
            return Number(op(self._value, other))

        # n * x
        if isinstance(other, Symbol):
            def _(**kwargs):
                a = kwargs[other._name]
                return op(self._value, a)
            return Functor(_)
```

...

```
        # n * f
        if isinstance(other, Functor):
            def _(**kwargs):
                return op(self._value, other(**kwargs))
            return Functor(_)

        assert False, "{} invalid other {}".format(self, other)
```

Testing partial arithmetic

```
>>> c = Variable('a')**2 + Variable('b')**2
>>> c
<__main__.Functor object at 0x103571b50>
>>> c(a=3, b=4)
25
>>> c(a=6, b=8)
100
>>> c = 2 * Variable('x') + Variable('y')
>>> c(x=2, y=3)
7
```

Recursive descent parser

E -> T { (‘+’ | ‘-’) E }
T -> F { (‘*’ | ‘/’) T }
F -> L [‘^’ F]
L -> *var* | *digit* | ‘(‘ E ‘)’ | ‘-’ T

- E is an expression
- T is a term
- F is a factor
- L is a literal

Implementing parser

```
import operator
Operator = {
   '+' : operator.add,
   '-' : operator.sub, …
}
def parseExpr(tokenizer):
   """
   implements E -> T { ( ‘+’ | ‘-’ ) E }
   """
   result = parseTerm(tokenizer)
   if accept(('+', '-'), tokenizer):
      apply = Operator[tokenizer.prevToken]
      expr1 = parseExpr(tokenizer)
      result = apply(result, expr1)

   return result
```

```
def parseTerm(tokenizer):
   """
   implements T -> F { ( ‘*’ | ‘/’ ) T }
   """
   result = parseFactor(tokenizer)
   if accept(('*', '/'), tokenizer):
      apply = Operator[tokenizer.prevToken]

      # {* /} T ...
      term = parseTerm(tokenizer)
      result = apply(result, term)

   return result
```

What does accept() do?

```
def accept(kind, tokenizer):
    # test if current token compares to expected
    if isinstance(tokenizer.currentToken, kind):

        # advance to the next token
        next(tokenizer)

        return True
    return False
```

- This example is incomplete but is enough to show the motivation.
- Why is it not complete?

Implementing parser (2)

```
def parseFactor(tokenizer):
    """
    implements F -> L [ '^' F ]
    """
    result = parseLiteral(tokenizer)

    if accept('^', tokenizer):
        apply = Operator[tokenizer.prevToken]

        # ^ factor
        factor = parseFactor(tokenizer)
        result = apply(result, factor)

    return result
```

```
def parseLiteral(tokenizer):
    """
    L -> var | digit | '(' E ')' | '-' T
    """
    if accept(Number, tokenizer):
        return tokenizer.prevToken

    if accept(Variable, tokenizer):
        return tokenizer.prevToken

    if accept('(', tokenizer):
        expr = parseExpr(tokenizer)
        if accept(')', tokenizer):
            return expr
```

What is tokenizer?

- Tokenizer is an object that: -
  - ❖Produce tokens of Number, Variable or raw but valid characters e.g. +, -, *, / for parser consumption hence the name tokenizer
  - ❖Simply tracks the current offset position as it advances to the next token, may implement Python iterator framework
  - ❖Keep current parsed token and previous token for comparison
  - ❖Skip uninteresting events such as space chars

Finally ...

```
>>> c = parse("a^2 + b^2")(a=3, b=4)
>>> print(c)
Number(25)
```

- Would you like to take a shot?

Think functionally

- Decompose a problem statement to functions
- Use functional language to express your problem
- Think about data transformation

Think recursively

Can you avoid loops?

```
def parseExpr(tokenizer):
    """
    implements E -> T { ( ‘+’ | ‘-’ ) E }
    """
    result = parseTerm(tokenizer)
    while accept(('+', '-'), tokenizer):
        apply = Operator[tokenizer.prevToken]
        term1 = parseTerm(tokenizer)
        result = apply(result, term1)
```

```
def parseExpr(tokenizer):
    """
    implements E -> T { ( ‘+’ | ‘-’ ) E }
    """
    result = parseTerm(tokenizer)
    if accept(('+', '-'), tokenizer):
        apply = Operator[tokenizer.prevToken]
        expr1 = parseExpr(tokenizer)
        result = apply(result, expr1)
```

Unit testing

Documentation