# Design Patterns in Python

By Steve Litt



Latest version available at:

http://www.troubleshooters.com/linux/presentations/golug_design_patterns_python/golug_design_patterns_python.pdf

**No Warranty: Use at your own risk.**

# Design Patterns You'll Learn

- By Reference
- Simple types
- Key/value bunch
- Arrays
- Functions
- Classes

- Branching
- Loops
- Subroutine
- Stack
- Queue

- Has-a
- Is-a
- Delegation
- Iterators
- Deepcopy
- Closures
- Functional pgm

# This Presentation Just Scratches The Surface

- Multitudinously more design patterns
  - Read more about design patterns
  - Read more about algorithms and data structures
- There are other ways to implement in these patterns in Python

# Python Vars Are References

- Not data locations, not data containers
- Just references to existing data. Labels.
- Run the following. Does changing b change a?

```
#!/usr/bin/python

a = 3
b = a
b = 4
print('a is {}, b is {}'.format(a, b))

a = ['zero', 'one', 'two', 'three']
b = a
b[1] = 'one and only'
print('a is {}, b is {}\n'.format(a, b))
```

# Deepcopy

- When you want identical but distinct copies
- Be careful, it can get confusing
- Python has deepcopy library

```
#!/usr/bin/python3
import copy
class Person:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
    def showname(self):
        print('{} {}'.format(
            self.fname, self.lname))

a = Person('Steve', 'Litt')
b = a
c = copy.deepcopy(a)
d = a
```

```
a.showname()
b.showname()
c.showname()
d.showname()

b.fname = 'Rena'

print('\n=======\n')
a.showname()
b.showname()
c.showname()
d.showname()
```

# Easy Stuff
# Look Up in Docs

- Simple Types
  - Numbers
  - Strings
- Builtin Composites
  - Key/Value bunches
    - Called dicts
    - Can represent anything
  - Arrays
  - Classes

# More Easy Stuff

- Functions (subroutines)
  - Functions are data (first class functions)
- Classes
- Branching
- Loops

# OOP's Not a Magic Bullet

- It makes some things easier, and some harder
- It's a tool, not a solution
- Many, many other design patterns
  - Functional programming
  - Callbacks
  - Closures
  - Iterators and generators
  - Procedural code
- Many, many more

# Pseudo OOP Requirements

- Encapsulating key/value bunch
- Functions are data
- Key/value pairs have reference to encapsulating key/value bunch
- Reasonable performance on the preceding requirements

# Stacks in Python

- Implemented as arrays
- Python arrays are indefinitely sized
- Stack methods
  - array.append()
  - array.pop()
  - len(array)

```
#!/usr/bin/python3
arr = ['one', 'two', 'three']
arr.append('four')
while len(arr) > 0:
    print(arr.pop())
```

# Queues (FIFOs) in Python

- Could be implemented with array
  - But that would be inefficent
- Implement as a collections.deque
  - Methods
    - queue.deque()
    - queue.append()
    - queue.popleft()
    - len(queue)

```
#!/usr/bin/python3
from collections import deque
queue = deque(['one',
    'two', 'three'])
queue.append('four')
while len(queue) > 0:
    print(queue.popleft())
```

# Data Centered Programming

- A philosophy more than a paradigm
- The data/algorithm tradeoff
- Data is more configurable and maintainable
- Data is a huge decision
  - Class, array, dict, stack, queue, tree, yaml, xml

# Has-a Relationships

- Implemented with dict or class
- Person “has a” last name

```
#!/usr/bin/python3
class Person:
    def __init__(self,
            fname,
            lname,
            job):
        self.fname = fname;
        self.lname = lname;
        self.job = job
person = Person('Jimmy',
    'Jones',
    'DBA');
print('{} {} is a {}.'.format(
    person.fname,
    person.lname,
    person.job
    ))
```

```
#!/usr/bin/python3
person = {'fname': 'Marty',
    'lname': 'Martinez',
    'job': 'Developer'}
print('{} {} is a {}.'.format(
    person['fname'],
    person['lname'],
    person['job']
    ))
```

# Is-a Relationships (Inheritance)

- Best when the relationship really is a subtype
- Best when adding specific behaviors or traits
- Benefits over Has-a:
  - Descendent inherits ancestor's methods
  - More intuitive and less work for subclass user
  - Traditional method for GUI programming
- Disadvantages compared to Has-a
  - More complex, especially with compilers
  - Less encapsulation

# Inheritance Example

- Square inherits Shape.move()

```
class Shape(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

```
class Square(Shape):
    def __init__(self, x, y, side):
        self.x = x
        self.y = y
        self.side = side
    def draw(self):
        st = 'nw=({},{}), ne=({},{}), se=({},{}), sw=({},{})'
        st = st.format(self.x, self.y,
                self.x + self.side, self.y,
                self.x + self.side, self.y + self.side,
                self.x, self.y + self.side)
        print(st)
```

```
mysquare = Square(1, 2, 3)
mysquare.draw()
print('==========')
mysquare.move(2, 4)
mysquare.draw()
```

# Delegation (Callbacks)

- Adds huge capabilities
- Perfect for events and sorts
- Reduce OOP reliance when OOP not right
- Make a generic algorithm specific
- Buzzword: First class functions
- See callbacks.py

# Closures

- Function within a function
- Inner function uses outer functions local vars
  - Those vars are called “up-values”
- >= Python 3x only
- Uses
  - Function factory
  - Pseudo OOP
  - Home-brew iterators and counters
  - Imagination's the limit

# Closure Example

```
#!/usr/bin/python3

def counter_factory():
    count = 0
    def counter():
        nonlocal count
        count = count + 1
        return count
    return counter

partctr = counter_factory()
chapctr = counter_factory()
sectctr = counter_factory()
```

```
print('Part {}, Part one'.format(str(partctr())))
print('Chapter {}, Chapter one'.format(str(chapctr())))
print('Section {}, Section one'.format(str(sectctr())))
print('Section {}, Section two'.format(str(sectctr())))
print('Section {}, Section three'.format(str(sectctr())))

sectctr = counter_factory() ## Restart section count
print('Chapter {}, Chapter Two'.format(str(chapctr())))
print('Section {}, Section one'.format(str(sectctr())))
print('Part {}, Part two'.format(str(partctr())))

sectctr = counter_factory() ## Restart section count
print('Chapter {}, Chapter Three'.format(str(chapctr())))
print('Section {}, Section one'.format(str(sectctr())))

sectctr = counter_factory() ## Restart section count
print('Chapter {}, Chapter Four'.format(str(chapctr())))
print('Section {}, Section one'.format(str(sectctr())))
print('Section {}, Section one'.format(str(sectctr())))
```

# Functional Programming

- Functions have no side effects
- Function return depends only on function args
- Little or no dependence on state
- Any state there is should be either
  - Internal to a function
  - Passed around through args and fcn returns
- Prefer recursion to iteration

# Functional Pgm: Tic-Tac-Toe

- Look how play() function defines the program
- Fcn side effects only for I/O
- Main logic: play() calls itself, swapping callbacks...
  - Recurses until a win or board filled
- Bad Steve: Iterative loops
- Bad Steve: Local variables
  - Local vars are more readable

# Tic-Tac-Toe: The Code

```
def play(current_symbol, apos, bpos, movea, moveb):
    display_board(current_symbol, apos, bpos)
    apos_new = movea(current_symbol, apos, bpos)
    if is_win(apos_new):
        display_board(current_symbol, apos_new, bpos)
        return current_symbol
    elif is_tie(apos_new, bpos):
        display_board(current_symbol, apos_new, bpos)
        return 'nobody'
    else:
        return play(swap_symbol(current_symbol),
                    bpos, apos_new,
                    moveb, movea)
```

# What You've Learned

- By Reference
- Simple types
- Key/value bunch
- Arrays
- Functions
- Classes

- Branching
- Loops
- Subroutine
- Stack
- Queue

- Has-a
- Is-a
- Delegation
- Iterators
- Deepcopy
- Closures
- Functional pgm

# Functional Pgm: Old Tic-Tac-Toe

- Fcn side effects only for I/O
- Main logic: human calls machine calls human...
  - Recurses until a win or board filled
- Bad Steve: board is huge state variable
  - But at least it's always an arg or fcn return
- Bad Steve: Iterative loops
- Bad Steve: Local variables
  - Local vars are more readable
- Bad Steve: No passing of functions