Object Oriented Programming in Python
Part 2

Jon Courtney

September 12, 2017

Afternoon Objectives

After this lecture, you should be familiar with. . .

- Basic Python decorators
- The *Callable* pattern
- Abstract Base Classes
- Verification, unit tests, and debugging

Decorators

A *decorator* is a function which wraps another function:

- Looks like the original function, i.e., `help(myfunc)` works correctly
- But, decorator code runs before and after decorated function
- Python provides some predefined decorators
- You can define your own decorators too

Common Python Decorators:

Some common decorators are:

- `@property` often with `@<NameOfYourProperty>.setter`
- `@classmethod` - can access class-wide state
- `@staticmethod` - groups functions under class namespace
- `@abstractmethod` - defines a method in an ABC
- Can also find decorators for logging, argument checking, and more

Properties

Properties look like member data:

- Actually returned by a function which has been decorated with `@property`
- Cannot modify the property unless you also create a setter, by decorating with `@<field_name>.setter`
- Gives you flexibility to change implementation later

@property Example

```python
class GasTank:

    # Initialize tank size and fuel
    def __init__(self, capacity=15):
        self.capacity = capacity
        self.fuel = 0        # Calls setter

    @property                # Getter
    def fuel(self):
        print('Checking the fuel gauge...')
        return self._fuel

    @fuel.setter             # Setter
    def fuel(self, gals):
        if gals > self.capacity:
            raise ValueError('Tank has overflowed!')
        else:
            self._fuel = gals
```

@classmethod Example

```python
class Math:
    pi = 3.14159265

    @classmethod
    def calc_area_circle(cls, r):
        return cls.pi * r**2


m = Math()
m.calc_area_circle(2.0)

# Also
Math.calc_area_circle(2.0)
```

@staticmethod Example

```python
class Math:
    pi = 3.14159265

    @classmethod
    def calc_area_circle(cls, r):
        return cls.pi * r**2

    @staticmethod
    def sqrt(n):
        return n**0.5

m = Math()
m.sqrt(4.0)

# Also
Math.sqrt(4.0)
```

Callable pattern

Class instances look & behave like a function but can hold state

- Implement `__call__` magic method
- Acts like a Functor in C++, i.e., like a function which can store state
- Often used with MapReduce because serializable and more flexible than a lambda or free function

Callable Example

```python
class MyCallable:
    def __init__(self, state):
        self.state = state

    def __call__(self, elem):
        '''Perform map operation on an element'''
        return elem**2

mc = MyCallable()              # Create the instance
results = map(mc, [1, 2, 3])   # Use like a function!
result  = mc(2)                 # Call like a function!
```

ABCs

An *Abstract Base Class* (ABC):

- Defines a standard interface for derived objects
- Cannot be instantiated – to ‘access,’ must derive a class from the ABC
- May contain some implementation for methods
- Can include abstract methods (with @abstractmethod decorator)

See doc on abc module for details

ABC Example

```python
from abc import ABC, abstractmethod

class Polygon(ABC):       # Can't create a Polygon object directly
    def __init__(self, vertices):
        self.vertices = vertices

    @abstractmethod
    def draw(self):       # Subclasses must override draw()
        pass

class Triangle(Polygon):
    def __init__(self, vertices):
        # Check that you have just 3!
        super().__init__(vertices)

    # Override draw() here:
    def draw(self):
        # Draw lines connecting points in self.vertices
        pass
```

Verification, unit tests, and debugging

Verification and debugging

Verifying your code is correct, and finding and fixing bugs are critical skills:

- Just because your code runs, doesn't mean it is correct
- Write unit tests to exercise your code:
  - Ensures interfaces satisfy their contracts
  - Exercise key paths through code
  - Identify any bugs introduced by future changes which break existing code
  - Test code before implementing entire program
- When unit tests fail, use a debugger to examine how code executes
- Both are critical skills and will save you hours of time

Unit tests and TDD

Unit tests exercise your code so you can test individual functions

- Use a unit test framework – **`unittest`**, `pytest` or `nose`
- Unit tests should exercise key cases and verify interface contracts
- A unit test can setup fixtures (i.e., resources) needed for testing
- *Test Driven Development* is a good approach to development:
    - *Red*: implement test and check it fails against stubbed code
    - *Green*: implement code (in KISS fashion) and verify it passes
        - "Premature optimization is the root of all evil" (Donald Knuth)
    - *Green*: refactor and optimize implementation
        - "Only refactor in the presence of working tests"
- Save time by verifying interfaces and catching errors early
- Catch errors if a future change breaks things

Design by Contract (DbC)

“Design by contract” is a helpful thought process when designing methods (and functions) and informs test-driven development.

Three components:

- **Preconditions**: What does the method/function expect when it begins?
- **Postconditions**: What does the method/function guarantee when it exits?
- **Invariants**: What must the method/function maintain for consistency?

Use `assert` to Test Preconditions

```python
class Triangle(Polygon):
    def __init__(self, vertices):
        # Verify preconditions
        assert len(vertices) == 3, "Not a triangle!"
        super().__init__(vertices)

    # Override draw() here:
    def draw(self):
        # Draw lines connecting points in self.vertices
        pass



t = Triangle([v1, v2])  # Throws AssertionError: Not a triangle!
```

Use try/except to Catch Problems Preemptively

```python
num = 1
denom = 0

try:
    ratio = num / denom
except ZeroDivisionError as err:
    print("To infinity and beyond!", err)
```

Using PDB

When unit tests fail, use the debugger to find a bug:

- If working in ipython, will display line of code which caused exception
- For complex bugs, debug via PDB
- To start PDB, at a specific point in your code, add:

  ```
  import pdb


  ...
  pdb.set_trace()  # Start debugger here
  ...
  ```

- See PDB's `help` for details
- Learn how to use a debugger. It will save you a lot of pain...

Essential debugging

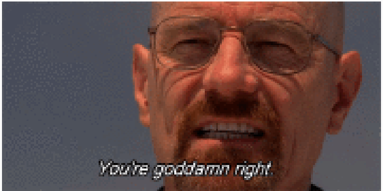
Once you have mastered one debugger, you have mastered them all:

Command	Action
`h`	help
`b`	set a break-point
`where`	show call stack
`s`	execute next line, stepping into functions
`n`	execute next line, step over functions
`c`	continue execution
`u`	move up one stack frame
`d`	move down one stack frame

Debugging tricks

Some hard-won debugging tips:

- When starting any project ask, ‘How will I debug this?’
- Program defensively: write code that anticipates problems
- If you cannot figure out what is wrong with your code, something you think is true most likely isn’t
- Explain your problem to a rubber duck . . . or friend
- Try to produce the smallest, reproducible test case
- If it used to work, ask yourself, ‘What changed?’
- Add logging, but beware of Heisenberg: when you measure a system, you perturb it . . .



Miscellaneous

*args and **kwargs

Shorthand to refer to a variable number of arguments:

- For regular arguments, use `*args`:
    - `*args` is a `list`
    - `def genius_func(*args):` to define a function which takes multiple arguments
    - Can also call function using a `list`, if you dereference

*args Example

- Case 1: supply all args via a list

```
def myargs(arg1, arg2, arg3):
    return arg1 * arg2 + arg3

>>> z = [ 2, 3, 4 ]
>>> myargs(*z)
10
```

- Case 2: process variable number of arguments

```
def args2list(*args):
    return [ix for ix in args]

>>> args2list(1, 2, 3, 4)
[1, 2, 3, 4]
```

*args and **kwargs (cont.)

- For keyword arguments, use **kwargs:
  - **kwargs is a dict
  - def genius_func(**kwargs): to define a function which takes multiple keyword arguments
  - Can also call function using a dict, if you dereference

```python
my_dict = {'a': 15, 'b': -92}
genius_func(**my_dict)
```

- Mixed order: some_function(args, *args, **kwargs)