Object-Oriented Programming Part 1

Prof. Pai H. Chou
National Tsing Hua University

Outline

- Objects
  - Object-oriented vs. procedure-oriented
  - Instantiation: copying vs. class
  - Example with Turtle Graphics
- Class definition
  - Constructor, methods
  - instance attributes vs. class attributes

What is an Object?

- Informally,
  - A general term for "things" in a program
  - anything that an identifier can reference
- More formally,
  - a unit of data that can "respond to messages"
  - in other words, bundled data and program code!

Objects in Python

- Values of built-in data types
  - `13`, `"hello"`, `22.5`, `['a', 'b', 'c']`, `{'x', 'y', 'z'}`, ...
- Why? because you can "send message" to them
  - "send message" in Python is to call a method on an object
  - e.g., `['a', 'b', 'c'].index('b')` returns `1`
    => `.index('b')` "sends a message" to the list object
    => "responds" by returning `1`
- in Python, types (classes) are also objects!
  - you can compare `if type(L) == str`
    (str is the name of a class!)

Two ways of making an object

- by copying
  - shallow copy vs. deep copy
- by instantiating from a class
  - send a "construct" message to a class
  - the class responds by ***instantiating*** an object
  - a ***constructor*** is a special method that initializes the newly created object

Copying object

- ```python
  import copy
  x = copy.copy(y)  # shallow copy
  x = copy.deepcopy(y) # deep copy
  ```
- Useful for mutable objects
  - immutable objects (int, float, str) don't need to be copied, because their values don't change!!!
- the source object is the "prototype"
  - behaves like the original, but different identity
  ```

Class-based object instantiation

- Class: definition, "blueprint"
  - member data and code (member function) that operates on data
- Instance: object created according to class

term	meaning
class	definition for data (attributes) and code (methods)
instance	an object created according to a class definition
instantiation	creation of an instance (based on a class)
method	a function defined in a class to operate on its data

Concept of data type

- In Python, a type is called a class
- Python provides built-in types
    - simple types: int, float, complex
    - str, list, tuple, set, dict, bytes
- Additional types can be defined
    - defined by modules
    - defined by users

Constructor call looks like function call!

- Constructor name = name of the type
- e.g., `s = set((1, 2, 3))`
  - looks like function call, but calls `set` constructor => creates an instance of `set` class, initialized based on parameter (1, 2, 3) => `{1, 2, 3}`
- similarly, `L = list('hello')`
  - calls constructor => `['h','e','l','l','o']`

Procedure-Oriented vs. Object-Oriented Programming

- Procedure-oriented:
  - Call procedures (i.e., functions) by passing objects as parameters
  - e.g.,
    ```
    L = [2, 5, 1, 7]
    M = sorted(L)
    ```
    `sorted()` is a function

- Object-oriented
  - Send messages to objects
  - Python: invoke methods
  - e.g.,
    ```
    L = [2, 5, 1, 7]
    L.sort()
    ```
    `.sort()` is a method

Reasons for Object Oriented

- Higher-level abstraction
  - "abstract data type" with some meaning (e.g., date, time, window, button, ...)
- keep code organized
  - list of methods = things you can do to the object
  - Decouple implementation from interface
- allow multiple instances
  - Instances do not interfere with each other

Example: turtle graphics

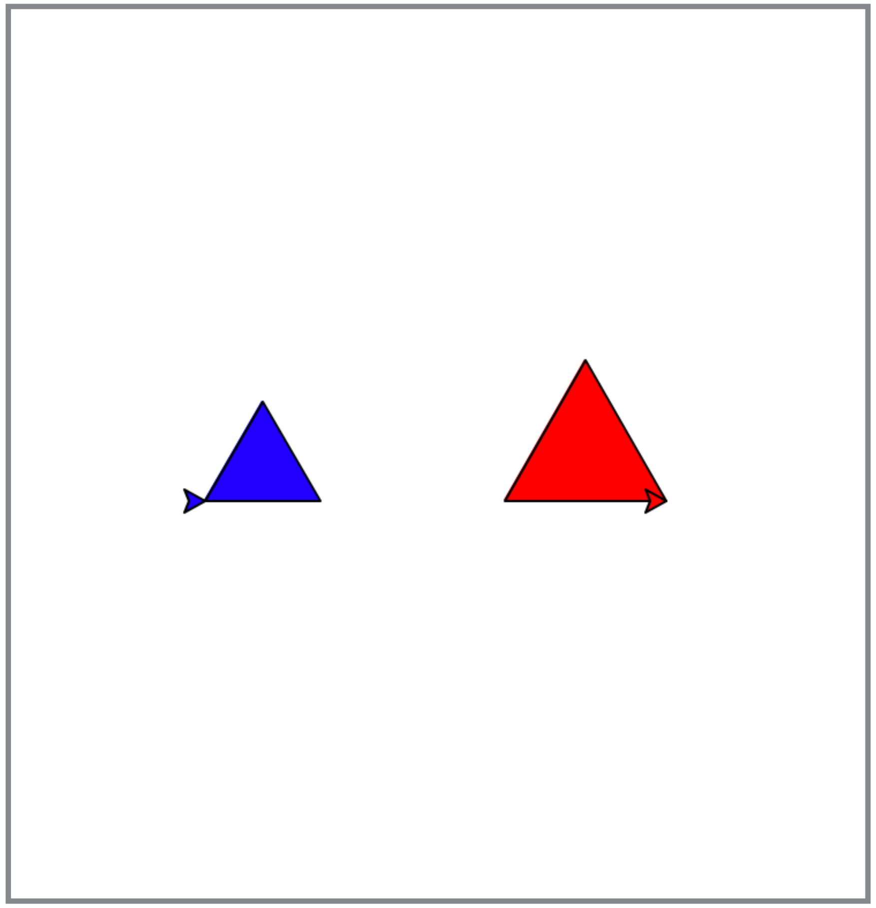
- classic way of drawing graphics using very simple commands
  - `import turtle     # import module`
  - `t1 = turtle.Turtle()   # instantiate Turtle`
  - `t2 = turtle.Turtle()   # instantiate another`
- allow drawing on a 2-D canvas in Cartesian coordinate
  - can up/down, hide/show turtle, goto, backward, forward, setheading, left, right, pencolor, dot

Methods of turtle graphics

purpose	API	
show/hide turtle	t.showturtle()	t.hideturtle()
pen up or down	t.up()	t.down()
turtle move by distance	t.forward(dist)	t.backward(d)
turtle turning #degrees	t.left(deg)	t.right(deg)
set turtle to coordinate & dir	t.goto(x,y)	t.setheading(deg)
get turtle coordinate & dir	t.pos()	t.heading()
set pen or fill color	t.pencolor(c)	t.fillcolor(c)
draw circle of diameter and color	t.dot(dia,c)	
bracket for filling polygon	t.begin_fill()	t.end_fill()
undo, clear what this turtle drew	t.undo()	t.clear()
print text	t.write(s, font, align)	

Example Turtle Graphics

```
>>> import turtle
>>> t1 = turtle.Turtle()
>>> t1.up()
>>> t1.goto(-100, 0); t1.down()
>>> t1.fillcolor('blue')
>>> t2 = turtle.Turtle()
>>> t2.up()
>>> t2.goto(100, 0); t2.down()
>>> t1.begin_fill()
>>> t2.fillcolor('red')
>>> t2.begin_fill()
>>> for i in range(3):
...     t1.forward(50)
...     t1.left(120)
...     t2.backward(70)
...     t2.right(120)
...
>>> t2.end_fill()
>>> t1.end_fill()
```

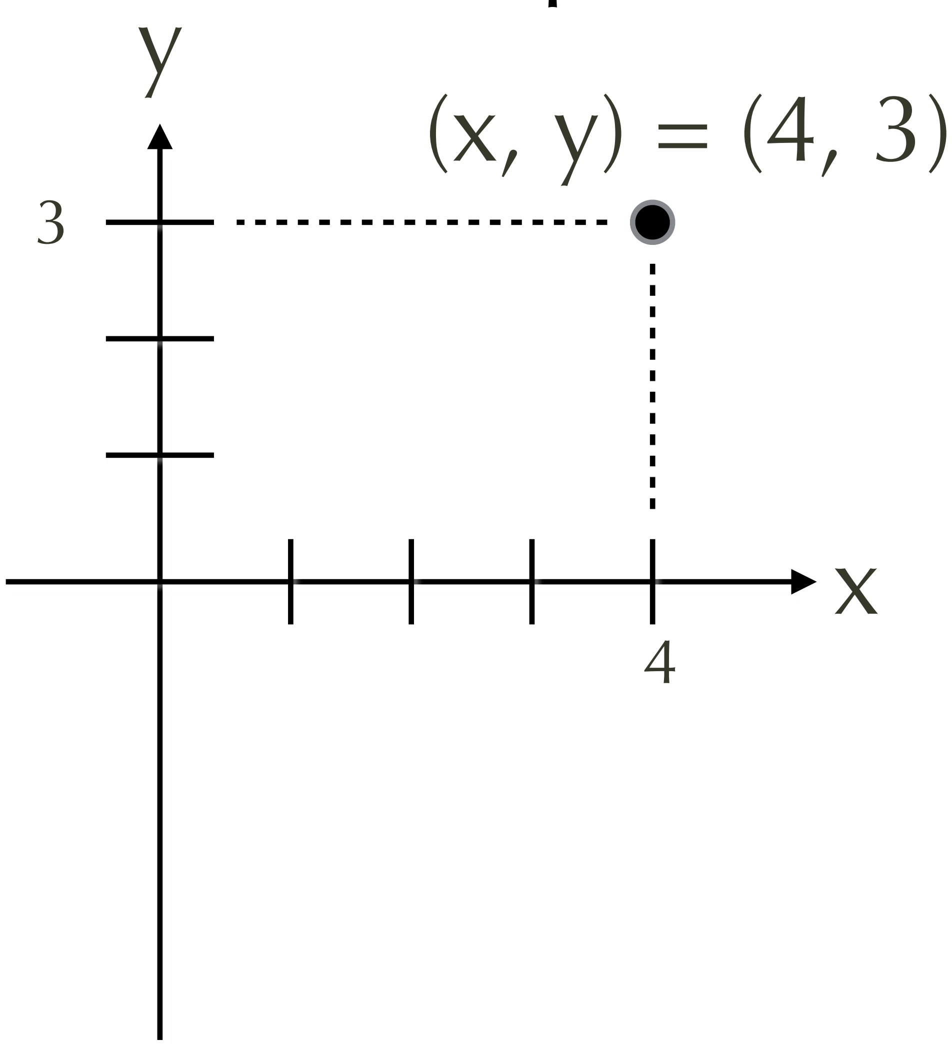


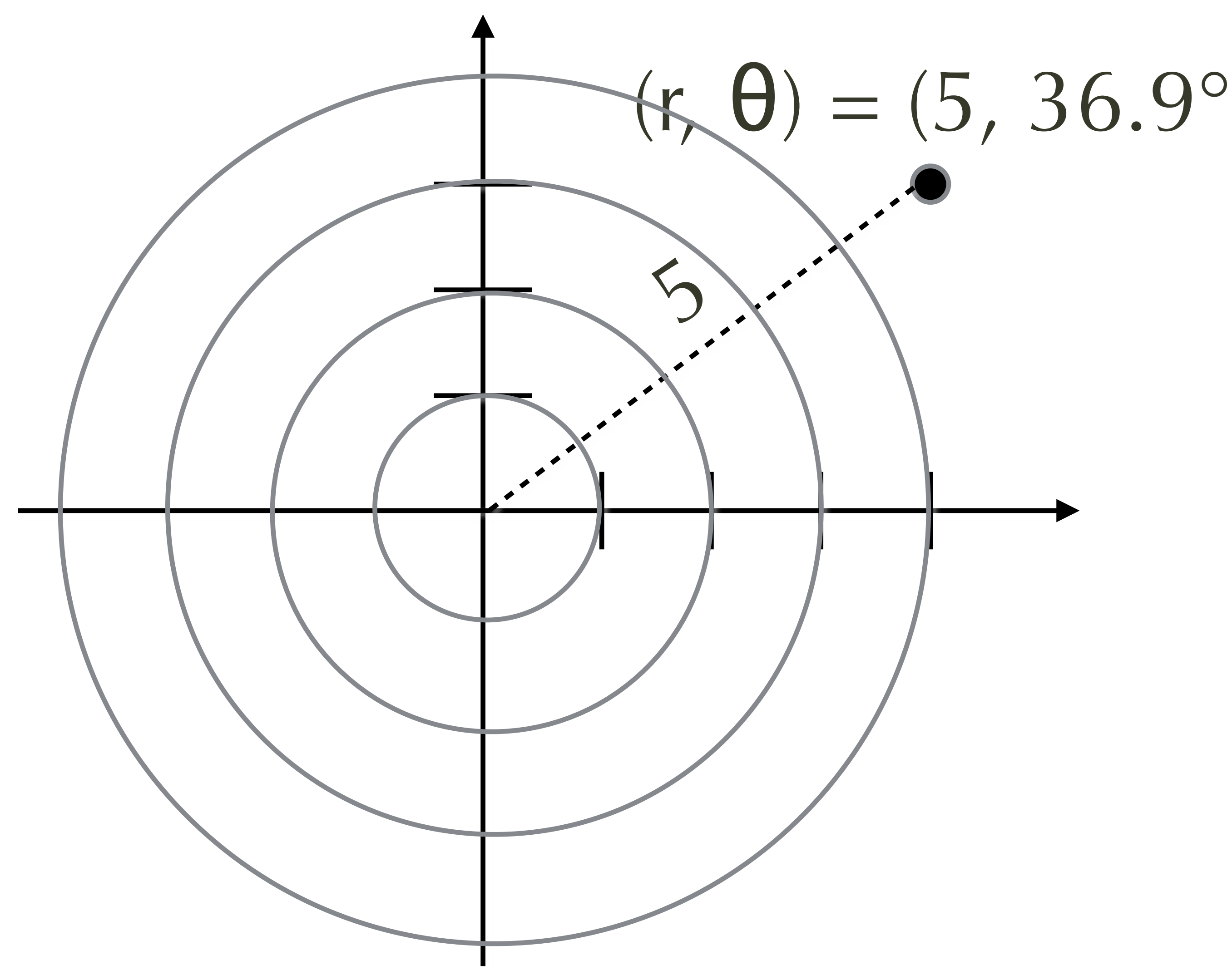
Defining your own class

- Purposes
  - higher-level concepts
  - separate representation from access format
  - impose constraints on allowable values
- What is needed
  - constructor, possibly with parameters
  - methods and attributes

Example: 2D Point class

- Multiple ways to represent a point in 2D
  - (x, y) in Cartesian coordinates, or complex #
  - (r, θ) in polar coordinates





Point as an Abstract data type

- (x, y) is just one way of representing a point
  - internally, it could also use (r, θ) => this should be of no concern to the user!
  - users just want a way to access it in the most convenient way => a method could do conversion
- Define a set of methods for Point

Example constructor for Point class

- Constructor is called upon instantiation
  - special name `__init__(self, args...)` within class

```
class Point:
    def __init__(self, x, y): # two underscores before and after
        self.x = x           # save param x as attribute of self
        self.y = y           # save param y as attribute of self
```

  - called as the *ClassName*(*args*) for instantiation

```
>>> p = Point(2, 3)  # instantiate a Point with (2, 3)
```

  - `p = Point(2, 3)`
    Python calls `Point.__init__(p, 2, 3)` to initialize `p`
  - formal param `self` refers to the instance being initialized (`p`)

Attributes of an object

- Also called "fields", "member data"
- Think of as variables local to object

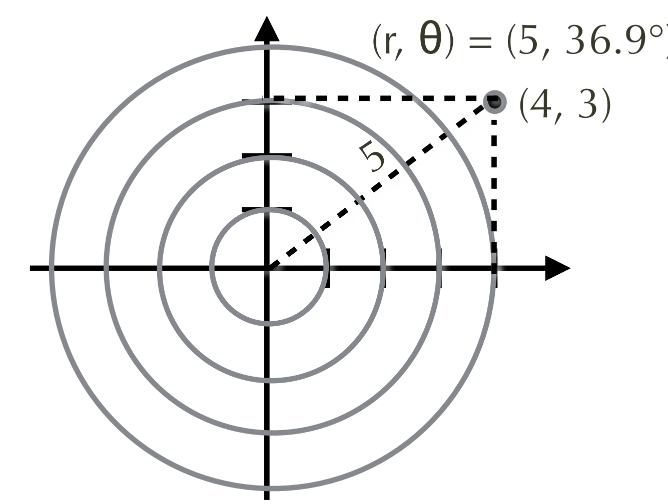
```
class Point:
    def __init__(self, x, y): # two underscores before and after
        self.x = x
        self.y = y

p = Point(2, 3)
```

- Constructor creates two attributes self.x and self.y based on parameters x and y
- instance p can access attributes as `p.x`, `p.y`

Derived attribute

- Those computable from other attributes
  - e.g., given (x, y) can calculate (r, θ)
  - define `r` and `theta` as methods




```
import math
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def r(self):
        return math.sqrt(self.x**2 \
                         + self.y**2)
        # = math.hypot(self.x, self.y)
    def theta(self):
        return math.atan(self.y / self.x)\
              * 180 / math.pi
```

```
>>> p = Point(4, 3)
>>> p.r()
5.0
>>> p.theta()
36.8698976458402
```

Object literal

- When typing in interactive mode,

```
>>> p = Point(2, 3)  # instantiate a Point with (2, 3)
>>> p
<__main__.Point object at 0x10f5c5400>
```

  - not very helpful...
  - Would be nice if it could show a "literal"!
- Solution: `__repr__` special method

__repr__ special method

- called by shell to get string to display

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return f'Point({self.x}, {self.y})'
```

  - it can be any string that is meaningful
  - most likely, looks like the constructor call!!

- So now Python shell can display it

```
>>> p = Point(2, 3)  # instantiate a Point with (2, 3)
>>> p
Point(2, 3)
>>>
```

Function operating on points

- example: move a point by dx, dy

```
>>> def proc_move_by(p, dx, dy):
...     p.x += dx
...     p.y += dy
...
>>> q = Point(2, 3)
>>> proc_move_by(q, 1, -2)
>>> q
Point(3, 1)
```

  - formal parameter p references the same Point object as q does

Method: member function

- Method is invoked on an instance

⚠ Python requires the first parameter of method to be `self`

method (inside class)

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move_by(self, dx, dy):
        self.x += dx
        self.y += dy
```

function (outside class)

```
def proc_move_by(p, dx, dy):
    p.x += dx
    p.y += dy
```

```
>>> q = Point(2, 3)
>>> proc_move_by(q, 1, -2)
>>> q
Point(3, 1)
```

```
>>> q = Point(2, 3)
>>> q.move_by(1, -2)
>>> q
Point(3, 1)
```

Methods for Point class

- Mutation
  - move_to(x, y) # absolute coordinate
  - move_by(dx, dy) # relative displacement
- Derived attributes
  - radius()
  - area_as_circle() # if treated as radius
  - area_as_rectangle # if treated as width and height

implementation for Point methods

```
import math
class Point:
    def __init__(self, x, y): ...
    def __repr__(self):       ...
    def r(self):              ...
    def theta(self):          ...

    def move_to(self, x, y):
        self.x = x
        self.y = y
    def move_by(self, dx, dy):
        self.x += dx
        self.y += dy
    radius = r
    def area_of_circle(self):
        return math.pi * self.r() **2
    def area_of_rectangle(self):
        return self.x * self.y
```

- All methods have `self` as first argument
- methods can access all attributes of its own class (`self`'s or another instance's)
- methods can call other methods
- `radius = r` declares `radius` to be an alias for `r`

@property for derived attributes

```
import math
class Point:
    def __init__(self, x, y): ...
    def __repr__(self):       ...
    @property
    def r(self):              ...
    @property
    def theta(self):          ...

    def move_to(self, x, y):
        self.x = x
        self.y = y
    def move_by(self, dx, dy):
        self.x += dx
        self.y += dy
    radius = r
    @property
    def area_of_circle(self):
        return math.pi * self.r **2
    @property
    def area_of_rectangle(self):
        return self.x * self.y
```

- Want to just name derived attributes without the calling () syntax
  - e.g., want to say p.radius instead of p.radius()
- Solution: use @property decorator in front of those methods
- ```
  >>> q = Point(4, 3)
  >>> q.r
  5.0
  ```
  ```

Advantages of @property decoration

- uniform syntax like regular attributes
  - object.attr , instead of () without passing arg
- can make attribute read-only access!
  - otherwise, other code can make data inconsistent

```
>>> q = Point(4, 3)
>>> q.radius
5.0
>>> q.radius = 20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Name spaces of class and instance

- Class
  - contains its own symbol table
  - Look at `Point.__dict__` or `dir(Point)`
  - class can also have its own attributes
- Instance
  - each instance has own symbol table, separate from class
  - constructor enters symbols into the instance's name space by self.x = value

Example name spaces of class and instance

```
>>> p = Point(2, 3)
>>> p.move_by
<bound method Point.move_by of <__main__.Point object at
0x10a862ac8>>
>>> dir(Point)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',...
'__subclasshook__', '__weakref__', 'move_by', ...]
>>> p.__dict__
{'x': 2, 'y': 3}
>>> p.__class__
<class '__main__.Point'>
>>> p.w = 'hello'     # entered into the instance's namespace
>>> Point.z = 'abc'   # entered into the class's namespace
>>> p.z               # instance can look into class's namespace
'abc'
>>> p.__dict__        # even though 'z' is not in instance's space
{'x': 2, 'y': 3}
```

Example use of Class Attributes

- Suppose we want to assign a unique serial number to each point that we create
  - need to keep a "global variable", but want it to be associated with the class
  - solution: class attribute

```
class Point:
    count = 0  # class attribute
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.serial = Point.count
        Point.count += 1
```

```
>>> p = Point(2, 3)
>>> q = Point(4, 5)
>>> p.serial  # p has serial no.0
0
>>> q.serial  # q has serial no.1
1
>>>
```

Class-specific Constraints on Value

- A class may limit allowed values of attributes
- example: ADT for date and time
  - possible attributes: year, month, day, hour, minute, second
  - day-of-week may be a derived attribute!
  - day-of-week is limited to Sun, Mon, ... Sat
  - month is limited to 1-12
  - day is limited to 1-28, 29, 30, or 31, depending on the year or month!!
- Q: How to enforce such constraints?
  Ans: by methods

Attribute getting vs. setting

- "get" = read an attribute's value
  - e.g., print(dt.hour) # read dt's hour attribute
- "set" = assign a new value to an attribute
  - e.g., dt.hour = 7   # set hour to 7 o'clock
- Q: How to ensure setting attribute is valid?
- A: Use "setter" method, and then decorate using @property

Example class: DateTime

- Suppose we want a constructor

  - ```
    >>> dt = DateTime(2019, 6, 30, 5, 20, 32)
    ```

    to construct the date and time for
    June 30, 2019 at 5:20:32 am

- is this a valid date and time? if not, need to raise an exception

  - ```
    >>> dt = DateTime(2019, 6, 31, 5, 20, 32)
    ValueError: day 31 out of range
    ```

Example: constructor for DateTime

- Original, unchecked

```
class DateTime:
    def __init__(self,year,month, \
        day, hour, minute, second):
        self._year   = year
        self._month  = month
        self._day    = day
        self._hour   = hour
        self._minute = minute
        self._second = second
```

- use _ to protect attribute from outside access

- Replace with "setter" methods

```
class DateTime:
    def __init__(self,year,month,\
        day, hour, minute, second):
         self.set_year(year)
         self.set_month(month)
         self.set_day(day)
         self.set_hour(hour)
         self.set_minute(minute)
         self.set_second(second)
```

- setter method checks before actually setting

How to write setter methods

```python
def check_range(field_name, field_value, L, U):
    if not (L <= field_value <= U):
        raise ValueError(f'{field_name} must be {L}..{U}')

class DateTime:
    def __init__(self, year, month, day, hour, minute, second):
        ...
    def set_year(self, year):
        if type(year) != int: raise TypeError('year must be int')
        self._year = year
    def set_month(self, month):
        check_range('month', month, 1, 12)
        self._month = month
    def set_day(self, day):
        if self._month in {1,3,5,7,8,10,12}:check_range('day',day,1,31)
        elif self._month in {4, 6, 9, 11}:  check_range('day',day,1,30)
        elif leap(self._year):              check_range('day',day,1,29)
        else:                               check_range('day',day,1,28)
        self._day = day
    def set_hour(self, hour):
        check_range('hour', hour, 0, 23)
        self._hour = hour
```

Boundary cases

- `set_day()` checks month and year => okay
- `set_month()` and `set_year()` are incomplete
  - suppose month=3, day=31, then `set_month(4)` => April does not have 31 days, should raise `ValueError` due to existing day setting!
  - similarly, month=2, day=29, leap year, then `set_year(2001)` non-leap year => cannot have 29 days! Should also raise `ValueError` due to 2/29
  - These extra checks need to be added to `set_month()` and `set_year()` access methods

Attribute access through __dict__

- obj.attr
  - dt._month
- obj.__dict__['attr']
  - dt.__dict__['_month']
  - dt.__dict__['_'+'month']

```
>>> dt = DateTime(2019, 6, 30, 5, 20, 32)
>>> dt._month
6
>>> dt.__dict__['_month']
6
>>> dt.__dict__['_month'] = 12
>>> dt._month
12
```

Q: Why access attribute through `__dict__`?

A: Because you can compute the key (attribute name)!

e.g., change from `'month'` to `'_month'`

Conversion to a helper setter

```python
class DateTime:
    ...
    def check_and_set(self, field_name, field_value, L, U):
        if not (L <= field_value <= U):
            raise ValueError(f'{field_name} must be {L}..{U}')
        self.__dict__['_'+field_name] = field_value
    def set_year(self, year):
        if type(year) != int: raise TypeError('year must be int')
        self._year = year
    def set_month(self, month):
        self.check_and_set('month', month, 1, 12)
    def set_day(self, day):
        if self._month in {1, 3, 5, 7, 8, 10, 12}:
            self.check_and_set('day', day, 1, 31)
        elif self._month in {4, 6, 9, 11}:
            self.check_and_set('day', day, 1, 30)
        elif leap(self._year):
            self.check_and_set('day', day, 1, 29)
        else:
            self.check_and_set('day', day, 1, 28)
    def set_hour(self, hour):
        self.check_and_set('hour', hour, 0, 23)
```

Example: getter and setter

```
class DateTime:
    def __init__(self, year, month, day, hour, minute, second):
        ...
    def get_month(self):
        return self._month
    def set_month(self, month):
        self.check_and_set('month', month, 1, 12)
```

- call getter/setter method to make changes

```
>>> dt = DateTime(2019, 6, 30, 5, 20, 32)
>>> dt.get_month()
5
>>> dt.set_month(8)
>>> dt.get_month()
8
>>> dt.set_month(13)
ValueError: day 31 out of range
```

How to package getter/setter as attribute access?

- getter method

```
>>> dt.get_month()
5
```

- attribute syntax

```
>>> dt.month
5
```

- setter method

```
>>> dt.set_month(8)
```

```
>>> dt.month = 8
```

- Solution: property

```
class DateTime:
    ...
    def get_month(self):
        ...
    def set_month(self, month):
        ...
    month = property(lambda self: self.get_month(), \
                     lambda self, v: self.set_month(v))
```

instance method, class method, and static method

- instance method (default)
  - implicit `self` as first parameter
- class method (`@classmethod`)
  - implicit `cls` as first parameter
- static method (`@staticmethod`)
  - no implicit first parameter; just like any other function but just scoped inside class

Class methods

- a method that is invoked on the class rather than on the instance
  - first argument is the class (cls), rather than the instance (self)
- Why? same reason as class attributes
  - most likely, want a method to protect access to a class attribute

Example: DateTime year range

- Previously, no limit on year, but may want user to set their own
  - e.g., limit valid year from -5000 to +4000
  - want getter/setter to view/set new range

```
class DateTime:
    year_from = -5000
    year_to = +4000
    def __init__(self, year, month, day, hour, minute, second):
        ...
    def set_year(self, year):
        self.check_and_set('year', year, \
                           self.year_from, self.year_to) # class attr
    def set_year_range(self, lower, upper):
        self.year_from, self.year_to = lower, upper
```

this won't work!

Solution: class method

- `@classmethod`
  - `cls` instead of `self`
  - can call it from either instance or class!

```
>>> dt = DateTime(2019,6,30,5,20,32)
>>> dt.year_from
-5000
>>> dt.set_year_range(-8000, +6000)
>>> dt.year_from
-8000
>>> DateTime.year_from
-8000
>>> DateTime.set_year_range(-100,3000)
>>> dt.year_to
3000
```

```
class DateTime:
    year_from = -5000
    year_to = +4000
      ...
    def set_year(self, year):
        self.check_and_set('year', year, \
                     self.year_from, self.year_to) # class attr
    @classmethod
    def set_year_range(cls, lower, upper):
        cls.year_from, cls.year_to = lower, upper
```

Static methods

- a method really just a function that is scoped in the class
  - there is no implicit argument(cls or self), unlike class method or instance method
- Why? it is just a function, not a method
  - most likely, want to provide some functions that are logically associated with the class but is independent of the instance or the class

Example: leap function in DateTime class

- leap(year) is leap-year test
  - What is the best way to structure it?
    (1) top-level function? (2) instance method (3) class method (4) static method?

```
class DateTime:
    def __init__(self, year, month, day, hour, minute, second):
        ...
    def set_day(self, day):
        if self._month in {1, 3, 5, 7, 8, 10, 12}:
            self.check_and_set('day', day, 1, 31)
        elif self._month in {4, 6, 9, 11}:
            self.check_and_set('day', day, 1, 30)
        elif leap(self._year):
            self.check_and_set('day', day, 1,2 9)
        else:
            self.check_and_set('day', day, 1, 28)
```

looks like a top-level function, but could be considered for a static method

Rewrite leap as static method in DateTime class

- can call leap() from either class or instance

```
class DateTime:
    def __init__(self, year, month, day, hour, minute, second):
        ...
    @staticmethod
    def leap(year):  # note: no self, no cls!!!
        return (year % 400 == 0) or \
               ((year % 4) == 0) and (year % 100 != 0)
    def set_day(self, day):
        if self._month in {1, 3, 5, 7, 8, 10, 12}:
            self.check_and_set('day', day, 1, 31)
        elif self._month in {4, 6, 9, 11}:
            self.check_and_set('day', day, 1, 30)
        elif self.leap(self._year):
            self.check_and_set('day', day, 1,2 9)
        else:
            self.check_and_set('day', day, 1, 28)
```

```
>>> dt = DateTime(...)
>>> dt.leap(2000)
True
>>> dt.leap(2001)
False
>>> DateTime.leap(2000)
True
>>> DateTime.leap(2001)
False
```

Example: leap function in four possible places

	defined as	calling form	Pro	Con

defined as	`def leap(year):`
	`    return ...`
	`class DateTime:`
		`leap(year)`	concise, no need for qualified name	name pollution

instance method	`class DateTime:`
	`  def leap(self,y):`
	`      ...`
		`self.leap(year)`	defined and called like any other method	self is passed but not used

class method	`class DateTime:`
	`  @classmethod`
	`  def leap(cls,y):`
	`      ...`
		`self.leap(y)`	defined similar to method	cls is passed but not used

static method	`class DateTime:`
	`  @staticmethod`
	`  def leap(y):`
	`      return ..`
		`self.leap(y)`	no name pollution, does not pass cls or self unnecessarily	

Summary of Part 1 OOP

- Object-oriented programming
  - Write code to defined data+code bundle
  - Send message to object; one way is method call
- Ways of creating objects
  - Duplication of prototype vs. instantiation of class
- Class-based OOP
  - Constructor, method definition, attribute
  - Value enforcement through get/set, property syntax