Classes II
Data Abstraction

- We will learn about how Python supports **data abstraction** (separating the data and details of the implementation from the user) via:
  - **Data hiding**: via attribute naming conventions (private, public)
  - **Encapsulation**: bundling together of data and methods that provide an interface to the data
Lecture Outline

- **Attribute types** (public/private) in Python
- **Print representation** via special method `__str__`
- **Accessor methods** and `@property`
- Putting it all together: `Coordinate` class.

![Coordinate class diagram](image-url)
Data Hiding Via Attribute Types
Attribute Naming Convention

- Double leading underscore (___) in name (strictly private): e.g. __val
  - Invisible from outside
  - Strong you cannot touch this policy
- Single leading underscore (_,) in name (private): e.g. _val
  - Can be accessed from outside, but shouldn’t
  - “Don’t touch this unless you are subclass”
- No leading underscore (public): e.g. val
  - Can be freely used outside class
- Conventions apply to procedural attributes (methods names) as well!
Attribute Naming Convention

```
In [1]: class TestingAttributes():
    __slots__ = ['__val', '_val', 'val']
    def __init__(self):
        self.__val = "I am strictly private."
        self._val = "I am private but accessible from outside."
        self.val = "I am public."

In [2]: a = TestingAttributes()

In [3]: a.__val

---------------------------------------------------------------------------
AttributeError                       Traceback (most recent call last)
<ipython-input-3-3e19e2bd1a2b> in <module>
    1 a.__val

AttributeError: 'TestingAttributes' object has no attribute '__val'

In [4]: a.val
Out[4]: 'I am private but accessible from outside.'

In [5]: a.val
Out[5]: 'I am public.'
```
__str__
Special method `__str__` is called when we print a class object

We can customize how the object is printed by writing a `__str__` method for our class

We can choose how the objects of the class are printed!

By default, if we print an object, it's not “pretty”
Defining the `__str__` method

class Coordinate(object):
    __slots__ = ['_x', '_y']
    def __init__(self, x, y):
        self._x = x
        self._y = y
    # other methods
    def __str__(self):
        return "<{}, {}>".format(self._x, self._y)

>>> print(pt)
<3, 4>
For Example: Name Class

```python
In [7]:
class Name:
    """Class to represent a person's name."""
    __slots__ = ['_f', '_m', '_l']

    def __init__(self, first, last, middle=' '):
        self._f = first
        self._m = middle
        self._l = last

    def __str__(self):
        if len(self._m):
            return '{}. {}. {}' .format(self._f[0], self._m[0], self._l)
        return '{}. {}' .format(self._f[0], self._l)

In [8]:
n1 = Name('Shikha', 'Singh')
n2 = Name('Iris', 'Howley', 'K.')

In [9]:
print(n1)
print(n2)

S. Singh
I. K. Howley
```
@property
Encapsulation is the bundling of data with the methods that operate on that data.

It is often accomplished by providing two kinds of procedural attributes:

- methods for retrieving or accessing the values of attributes, called **getter methods** or **accessor methods**. Getter methods do not change the values of attributes, they just return the values, and

- methods used for changing the values of attributes, called **setter methods**.
Accessor Methods via `@property`

- **Annotations @.** Python provides a rich collection of syntactic notes that can change how code is interpreted, called annotations.

- These are typically prefixed with the at-sign (@).

- **Accessor methods** do not change the state of the calling object and are used just to retrieve some information about the object.

- **@property annotation.** Treat a procedural attribute as a data attribute:
  - If we’d like to treat an accessor method as-if it were a data attribute, we can use the `@property` annotation
Back to the Coordinate Class
Euclidean Distance

\[ d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \]
Coordinate Class

- Use the `class` keyword to define a new type

```python
class Coordinate(object):
    # define attributes here
    # indented body of class definition
```

- the word `object` means `Coordinate` is a Python `object` and inherits all its attributes
  (inheritance will be covered in later lectures)

- `Coordinate` is a subclass of `object`
- `object` is a superclass of `Coordinate`
Initializing the Class: __init__

- Recall __init__ lets us initialize some data attributes of the class
- Recall __slots__ stores the data attribute names as strings in a list
- Single leading underscore signals private data or procedural attribute

```python
class Coordinate(object):
    __slots__ = ['_x', '_y']
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Can assign values to an instance of a class using dot notation.

Single leading underscore: private data attributes

Parameter to refer to an instance of the class
class Coordinate(object):
    """Represents the coordinates of a point."""
    __slots__ = ['_x', '_y']
    def __init__(self, x, y):
        self._x = x
        self._y = y

@property
def x(self):
    return self._x

@property
def y(self):
    return self._y

def _subX(self, other):
    """Subtracts the x coordinates of self and other and returns the answer"""
    return self._x - other._x

def _subY(self, other):
    """Subtracts the y coordinates of self and other and returns the answer"""
    return self._y - other._y

def dist(self, other):
    sqX = self._subX(other)**2
    sqY = self._subY(other)**2
    return round((sqX + sqY)**0.5, 2)

@property
def radius(self):
    """Returns the distance of the point from (0,0)"""
    origin = Coordinate(0, 0)
    return self.dist(origin)

def __str__(self):
    return '<{{}, {}}}'.format(self._x, self._y)
Acknowledgments

These slides have been adapted from:

- [http://cs111.wellesley.edu/spring19](http://cs111.wellesley.edu/spring19) and
- [https://www.python-course.eu/python3_object_oriented_programming.php](https://www.python-course.eu/python3_object_oriented_programming.php)