Introduction to Classes
Objects

- Python supports many different kinds of data:
  - 1234
  - 3.14159
  - "Hello"
  - [1, 5, 7, 11, 13]
  - {"CA": "California", "MA": "Massachusetts"}

- Each of these is an **object**, and every object has:
  - a **type**
  - an internal **data representation**
  - a set of functions for **interaction** with the object

- An object is an **instance** of a type:
  - 1234 is an instance of an **int**
  - "hello" is an instance of a **string**
Python is an “object-oriented” language

Question. What is an object?

Objects are a data abstraction that capture:

- An internal representation (through data attributes)
- An interface for interacting with the object
  - through methods (aka procedures /functions)
  - defines behavior but hides implementation

```
>>> def greeting():
    ...     print("Hello")
    ...
>>> type(greeting)
<class 'function'>
```
Example: \([1,2,3,4]\) has type \texttt{List}

- Lists are represented internally by a sequence of cells connected via pointers (called linked list)

\[
L = \begin{array}{c}
1 \rightarrow \quad 2 \rightarrow \quad 3 \rightarrow \quad 4 \\
\end{array}
\]

- This representation is \textit{private}
  - The user doesn’t need to know it to use list object
- How do manipulate lists? (interface through methods)
  - \texttt{L.append()}, \texttt{L.extend()}, \texttt{etc.}

**Summary.**
- Internal representation of objects should be private.
- Objects are manipulated through associated methods/ functions.
Creating Our Own Types: Classes

- We can create our own type by **defining our own class**

- Creating a class involves
  - Defining the **class name and its attributes**
    - E.g., someone wrote the code to implement a **list class**
  - Using the class involves
    - Creating **new instances** (objects)
      - E.g., `L = list()`
    - Doing operations on the instances
      - E.g., `L.append(3)`
Defining Our Own Type: Book class

Name of class (convention capital first letter)

```python
class Book(object):
    
    """This class represents a book""

    # define attributes here

    # indented body of class definition
```

Optional parent class

- Creating an instance of the class:

```python
b1 = Book()
```

Object/instance of class Book
Data Attributes or Instance Variables

- Objects have “state,” which is typically held in instance variables or (a very Pythonic terms:) attributes.
- For example, an object of class Book may have attributes like the name of the book and its author.
- We could assign these attributes directly to an instance of the class but we should never do this.

```python
b1 = Book()
b1.name = “Emma”
b1.author = “Jane Austen”
```

Attributes should typically not be be assigned outside class definition
Classes: Methods
Methods or Procedural Attributes

- Think of methods as object-specific functions
- They are defined as part of the class definition and describe how to interact with the class objects
- Example, methods for the list class

```
In [1]: L = list()
In [2]: L.extend([1,2,3])
In [3]: L
Out[3]: [1, 2, 3]
In [4]: L.append(4)
In [5]: L
Out[5]: [1, 2, 3, 4]
```
Our First Method

• How do we call the greeting?

• We create an instance of the class and call the method on that instance using the dot operator as follows:

```python
In [1]: class A:
    
    """Class to test the use of methods""
    
    def greeting(self):
        print("Hello")

In [2]: a = A()

In [3]: a.greeting()

Hello
```
Understanding Method Calls

• The following two calls are equivalent:

```python
In [1]: class A:
    """Class to test the use of methods""
    def greeting(self):
        print("Hello")
```

\[
a = A()
\]

\[
a.greeting()  # method 1
\]

\[
A.greeting(a)  # method 2
\]

Preferred/Standard way
self Parameter

- Even though method definitions have `self` as the first parameter (and we use this variable inside the method body), we don’t pass this parameter explicitly.

- This is because whenever we call a method on an object, the object itself is passed as the first parameter.

- Methods are object specific-functions and this lets us access the object’s properties via the methods directly.

- In some languages this parameter is implicit but in Python it is explicit and by convention named `self`.
Summary of Methods

• A method differs from a function only in two aspects:
  • It **belongs to a class**, and it is defined within a class
  • Its purpose is to provide an interface to access/manipulate objects
  • The first parameter in the definition of a method attribute is **the reference to the calling instance**.
  • This parameter that references the calling object is (by convention) called "**self**".
Initializing a Class: `__init__`

- While Python allows you to assign attributes to instances of a class on the fly (and outside the class), it is not the proper way to do so.
- You should never assign or modify attributes of an object manually.
- Data attributes should be initialized as part of the class definition.
- We can achieve this by the Python's special method `__init__`.
- `__init__`: Special method that lets us define how to create an instance of a class, by initializing some data attributes.

```python
class Book:
    """This class represents a book""

    def __init__(self, name=None, author=None):
        self.name = name
        self.author = author
```
__slots__
Avoid Dynamically Created Attributes

- Attributes of objects are stored in a dictionary `__dict__`
- Like any other dictionary, you can add items to `__dict__` on the fly and there are no predetermined set of keys
- This is why we can dynamically add attributes to objects (even though this is not recommended)

```python
In [6]: class Book:
    """This class represents a book""
    def __init__(self, name=\'None\', author=\'None\_\):  
        self.name = name  
        self.author = author

In [7]: newBook = Book(\'Emma\', \'Jane Austen\')

In [8]: newBook.__dict__
Out[8]: {\'name\': \'Emma\', \'author\': \'Jane Austen\'}
```
Avoid Dynamically Created Attributes

- Attributes of objects are stored in a dictionary `__dict__`
- Like any other dictionary, you can add items to `__dict__` on the fly and there are no predetermined set of keys
- This is why we can dynamically add attributes to objects (even though this is not recommended)

```python
In [7]: newBook = Book('Emma', 'Jane Austen')
In [8]: newBook.__dict__
Out[8]: {'name': 'Emma', 'author': 'Jane Austen'}
In [9]: newBook.year = 1815
In [10]: newBook.__dict__
Out[10]: {'name': 'Emma', 'author': 'Jane Austen', 'year': 1815}
```
__slots__

- Dynamic creation and assignment of attributes is not desirable
- Slots provide a clean way to avoid this: instead of having a dynamic dict that stores the attributes as (key, value) pairs, slots provide a static structure which prohibits addition of attributes

```python
In [18]: class Book:
    """This class represents a book""
    __slots__ = ['name', 'author']
    def __init__(self, name=None, author=None):
        self.name = name
        self.author = author

In [20]: b = Book('Emma', 'Jane Austen')

In [21]: b.year = 1815

---------------------------------------------------------------------------
AttributeError                       Traceback (most recent call last)
<ipython-input-21-58a49885b6e1> in <module>
----> 1 b.year = 1815

AttributeError: 'Book' object has no attribute 'year'
```
More Methods for the Book Class
Methods and Data Abstraction

• Methods of a class typically fall into two categories
  • **accessor methods** (that give us ready-only access to the object’s attributes)
  • **mutator methods** (that let us modify the object’s attributes)
• Ideally, we do not allow the user direct access to the object’s attributes
• Instead we control access to state through methods
• This approach enforces **data abstraction**
  • Methods provide a public interface
  • Attributes are part of the private implementation
Defining More Methods

- We define the following methods in the class definition of `Book` to provide an interface to our book objects:
  - `numWordsName` that returns the number of words in the name of the book
  - `sameAuthorAs` that takes another book object as parameter and checks if the two books have the same author or not
  - `yearSincePub` that takes in the current year and returns the number of years since the book was published
- Find the implementation and invocations of these methods in the Jupyter Notebook for the lecture.
Acknowledgments

These slides have been adapted from:

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