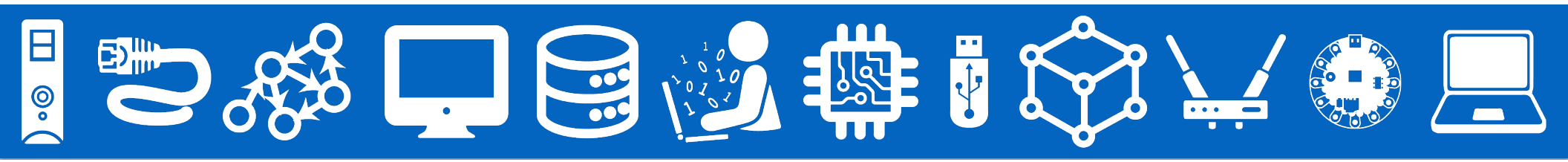
CS134:
Classes & Objects 2



Announcements & Logistics

- **Lab 8** is going to be a **partner lab**
  - **Must attend one lab session together**
  - Mon lab due on Wed, Tue lab due on Thur
- **Lab 6** graded feedback: coming soon (sorry for the delay)
- **HW 7** due Mon 10 pm (fewer questions this week)

Do You Have Any Questions?

Last Time

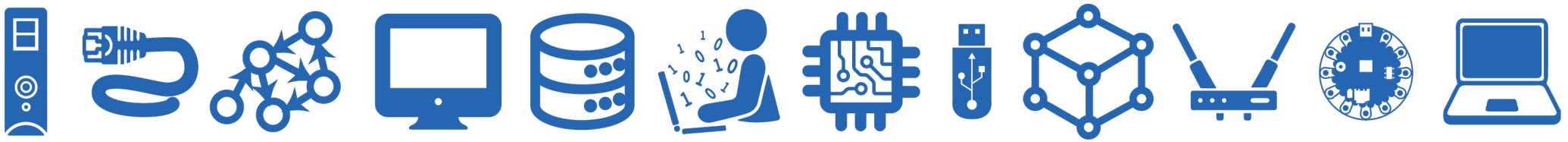
- Introduced the big idea of **object oriented programming** (OOP)
- Everything in Python is an object and has a type!
  - We can create **classes** to define our own types
- Learned how to define and call **methods** on objects of a **class**
  - Methods facilitate **abstraction:** hide unnecessary details
  - Discussed using the `self` parameter in methods of a class (`self` is a a reference to the calling instance)

- Quick aside: **functions versus methods**?
  - Functions are not associated with a specific class
  - Methods are associated with a specific class and are invoked on instances of the class (using dot notation)

Today's Plan

- Implement a simple Book class and learn about the following:
  - Declaring data **attributes** of objects using `__slots__`
  - Learning about scope and naming conventions in Python
  - Using the `__init__()` method to initialize objects with their attribute values
  - Defining **accessor** and **mutator** methods to interact with attributes
  - Implementing and invoking methods in general
  - Implementing `__str__()` method to provide meaningful print statements for custom objects

Defining a Class



Recap: Defining a Class

- Key features of a class:
  - **Attributes** that describe instance-specific data
  - **Methods** that act on those attributes
- When defining a new class (aka an object blueprint), we identify what attributes are required and what actions will be performed
- For example, suppose we want to define a new Book class
  - Attributes?
    - Title, author, publication year, genre, …
  - Methods?
    - sameAuthorAs( ), yearsSincePub( ), …

Recap: Attributes

- Objects have *state* which is typically held in **attributes**
  - For our **Book** class, these include the book's title, author, and publication year
  - Every **Book** instance has different attribute *values*!
- In Python, we **declare** attributes using `__slots__`
- `__slots__` is a **list of strings** that stores the **names** of all attributes in our class (only names of attributes are stored, not the values)
- `__slots__` is typically defined at the top of our class (before method definitions)

Declaring Attributes in `__slots__`

```
class Book:
    """This class represents a book"""
    # declare Book attributes
    __slots__ = ["_author", "_title", "_year"]


    # indented body of class definition
```

"_author", "_title", and "_year" are protected **attributes** of the Book class

Notice the use of _

Scope & Naming Conventions in Python

- Double leading underscore (__) in name (**strictly private**): e.g. `__value`
  - "Invisible" from outside of the class
  - Strong ***"you cannot touch this"*** policy (which is enforced)
- Single leading underscore (_) in name (**private/protected**): e.g. `_value`
  - Can be accessed from outside, but really shouldn't
  - ***"Don't touch this (unless you are a subclass)"*** policy
  - **Most attributes in CS134 should start with a single underscore**
- No leading underscore (**public**): e.g. `value`
  - Can be freely used outside class
- These conventions apply to **methods names** and **attributes**

Attribute Naming Conventions

```
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."
```

```
>>> a = TestingAttributes()
>>> a.__val
AttributeError: 'TestingAttributes' object has no attribute '__val'
>>> a._val
'I am private but accessible from outside.'
>>> a.val
'I am public.'
```

Note: Although we can access attributes directly using dot notation, it's bad practice. We should **always** use methods to manipulate attributes. More on this later.

Initializing a Class: `__init__`

- How do we assign **values** to the **attributes** in `__slots__`?
- Attributes should be assigned initial values when creating new instances
- We can achieve this using the special `__init__` method in Python
  - The initializer method, like a constructor in Java
- The `__init__` method is **run automatically anytime a new instance of a class is created**

```
class TestInit:
    """ This class will test when __init__ is called"""
    def __init__(self):
        print("__init__ is called")
```

```
>>> obj = TestInit()
__init__ is called
```

Book class: `__init__`

- The `__init__` method should set values for attributes in `__slots__`
- Values are often provided as parameters to `__init__`

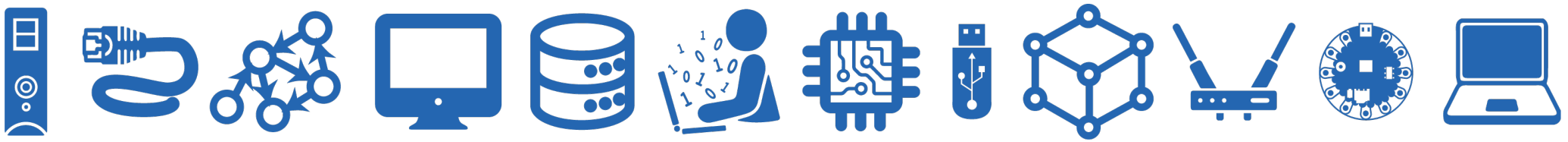
```
class Book:
     """This class represents a book with attributes title, author, and year"""
     # attributes
     # _ indicate that they are protected
     __slots__ = ["_title", "_author", "_year"]
     def __init__(self, bookTitle, bookAuthor, bookYear):
          self._title = bookTitle
          self._author = bookAuthor
          self._year = bookYear
```

When referring to class attributes, we use `self.{attribute name}`.

```
>>> # creating book objects
>>> pp = Book("Pride and Prejudice", "Jane Austen", 1813)
>>> emma = Book("Emma", "Jane Austen", 1815)
>>> ps = Book("Parable of the Sower", "Octavia Butler", 1993)
>>> ps._title
'Parable of the Sower'
```

Once again, it is better to use a method to access attributes. We'll fix this soon!

Class Methods



Default Argument Values

- We can provide default argument values in method and function definitions

```
class Book:
    """This class represents a book with attributes title, author, and year"""
    # attributes
    __slots__ = ["_title", "_author", "_year"]
    def __init__(self, bookTitle='', bookAuthor='', bookYear=0):
        self._title = bookTitle
        self._author = bookAuthor
        self._year = bookYear
```

- If we create a Book and don't provide values for the arguments in `__init__`, the values are set to be the default values ("" and 0 in this case)

```
>>> emptyBook = Book()
>>> emptyBook._title
''
```

Methods and Data Abstraction

- Ideally, we should not allow direct access to the object's attributes:

```
>>> # creating book objects
>>> ps = Book("Parable of the Sower", "Octavia Butler", 1993)
>>> ps._title
'Parable of the Sower'
```

- Instead we control access to attributes through accessor and mutator methods and avoid accessing the attributes directly
    - **Accessor methods:** provide "read-only" access to the object's attributes ("**getter**" methods)
    - **Mutator methods:** let us modify the object's attribute values ("**setter**" methods)
- This is called **encapsulation**: the bundling of data with the methods that operate on that data (another OOP principle)

```
class Book:
    """This class represents a book with attributes title, author, and year"""

    # attributes
    __slots__ = ['_title', '_author', '_year']

    # __init__ is automatically called when we create new Book objects
    # we set the initial values of our attributes in __init__
    def __init__(self, bookTitle='', bookAuthor='', bookYear=0):
        self._title = bookTitle
        self._author = bookAuthor
        self._year = int(bookYear)

    # accessor (getter) methods
    def getTitle(self):
        return self._title

    def getAuthor(self):
        return self._author

    def getYear(self):
        return self._year

    # mutator (setter) methods
    def setTitle(self, bookTitle):
        self._title = bookTitle

    def setAuthor(self, bookAuthor):
        self._author = bookAuthor

    def setYear(self, bookYear):
        self._year = int(bookYear)
```

Accessor methods return values of attributes, but do not change them

```python
class Book:
    """This class represents a book with attributes title, author, and year"""

    # attributes
    __slots__ = ['_title', '_author', '_year']

    # __init__ is automatically called when we create new Book objects
    # we set the initial values of our attributes in __init__
    def __init__(self, bookTitle='', bookAuthor='', bookYear=0):
        self._title = bookTitle
        self._author = bookAuthor
        self._year = int(bookYear)

    # accessor (getter) methods
    def getTitle(self):
        return self._title

    def getAuthor(self):
        return self._author

    def getYear(self):
        return self._year

    # mutator (setter) methods
    def setTitle(self, bookTitle):
        self._title = bookTitle

    def setAuthor(self, bookAuthor):
        self._author = bookAuthor

    def setYear(self, bookYear):
        self._year = int(bookYear)
```

Mutator methods change the value of attributes but do not explicitly return anything

Using Accessor/Mutator Methods

```
>>> pp.getTitle()
'Pride and Prejudice'
>>> emma.getAuthor()
'Jane Austen'
>>> ps.getYear()
1993
>>> ps.setYear(1991)
>>> ps.getYear()
1991
```

Use accessor methods to get the values of the attributes (when outside of class implementation)

Use mutator methods to set or change the values of the attributes (when outside of class implementation)

Defining More Methods

- Beyond the accessor and mutator methods, we can define other methods in the class definition of `Book` to manipulate or answer questions about our book objects:
    - `numWordsInTitle():` returns the number of words in the title of the book
    - `yearSincePub(currentYear):` takes in the current year and returns the number of years since the book was published
    - `sameAuthorAs(otherBook):` takes another Book object as a parameter and checks if the two books have the same author

```python
class Book:
    """This class represents a book with attributes title, author, and year"""

    # attributes
    __slots__ = ['_title', '_author', '_year']

    # __init__ is automatically called when we create new Book objects
    # we set the initial values of our attributes in __init__
    def __init__(self, bookTitle='', bookAuthor='', bookYear=0):
        self._title = bookTitle
        self._author = bookAuthor
        self._year = int(bookYear)

    # accessor (getter) methods
    def getTitle(self):
        return self._title

    def getAuthor(self):
        return self._author

    def getYear(self):
        return self._year

    # mutator (setter) methods
    def setTitle(self, bookTitle):
        self._title = bookTitle

    def setAuthor(self, bookAuthor):
        self._author = bookAuthor

    def setYear(self, bookYear):
        self._year = int(bookYear)

    # methods for manipulating Books
    def numWordsInTitle(self):
        """Returns the number of words in title of book"""
        return len(self._title.split())

    def sameAuthorAs(self, otherBook):
        """Check if self and otherBook have same author"""
        return self._author == otherBook.getAuthor()

    def yearsSincePub(self, currentYear):
        """Returns the number of years since book was published"""
        return currentYear - self._year
```

Invoking Class Methods

- We invoke methods on specific instances of our class
- In this example, we are invoking Book methods on specific Book objects

```
>>> # creating book objects
>>> pp = Book("Pride and Prejudice", "Jane Austen", 1813)
>>> emma = Book("Emma", "Jane Austen", 1815)
>>> ps = Book("Parable of the Sower", "Octavia Butler", 1993)
>>> ps.numWordsInTitle()
4
>>> emma.YearsSincePub(2022)
207
>>> ps.YearsSincePub(2022)
29
>>> ps.sameAuthorAs(emma)
False
>>> emma.sameAuthorAs(pp)
True
```

Print Representation of an Object

```
class TestingPrint():
   __slots__ = ["_attr"]

   def __init__(self, value):
      self._attr = value
```

```
>>> test = TestingPrint("testing")
>>> print(test)
<__main__.TestingPrint object at 0x105eecca0>
```

By default, if we print an object, the output is not helpful

- Special method **`__str__`** is automatically called when we ask to print a class object in Python
- **`__str__`** must always return a string
- We can customize how the object is printed by writing a custom **`__str__`** method for our class
- *Very* useful for debugging!

`__str__` for Book class

- What is a useful string representation of a **Book**?
  - Something that combines the attributes in a meaningful way
  - The `format( )` string method comes in handy here

```
# __str__ is used to generate a meaningful string representation for Book objects
# __str__ is automatically called when we ask to print() a Book object
def __str__(self):
   return "'{}', by {}, in {}".format(self._title, self._author, self._year)
```

  - Now when we ask to print a specific instance of a **Book**, we get something useful

```
>>> print(emma)

'Emma', by Jane Austen, in 1815
```

Summary

- Today we built a simple `Book` class
- **Declared attributes** using `__slots__`
- (Briefly) Learned about about scope and naming conventions in Python
- Used the `__init__()` method to initialize Book objects with their attribute values
- Defined **accessor** and **mutator** methods to interact with attributes and avoid accessing attributes directly
  - Note about mutators: If an attribute should not change, no need to define a setter method for it!
- Implemented a few more "interesting" Book methods
- Implemented the `__str__()` method so that we get meaningful print statements for our Book objects

The end!