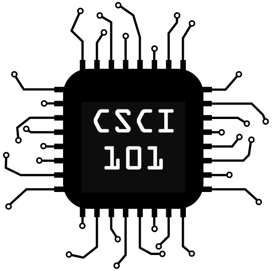
Object Oriented Programming

Making Your Own Data Types



Classes

A simple class can be defined like so:

```python
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

Classes

A simple class can be defined like so:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

A few things to notice:

- `__init__` is the initializes the object. It's actually what is called a **magic method**
- All the methods of the class take a parameter `self`, the object you are working on

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.
- `__lt__`, `__eq__`, etc. allow you to define comparisons.

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.
- `__lt__`, `__eq__`, etc. allow you to define comparisons.
- `__len__` binds into Python's `len(·)`

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.
- `__lt__`, `__eq__`, etc. allow you to define comparisons.
- `__len__` binds into Python's `len`$(\cdot)$
- There's far more than I can mention here. Read the docs!

Magic Methods

Magic methods are methods with certain names that allow you to bind features of your class to certain Python features.

- `__init__` was the simple example we just saw.
- `__del__` gets called when your object gets destructed.
- `__lt__`, `__eq__`, etc. allow you to define comparisons.
- `__len__` binds into Python's `len(·)`
- There's far more than I can mention here. Read the docs!

Why do this rather than `.equals()`, `.length()` and such?

In the face of ambiguity, refuse the temptation to guess. There should be one – and preferably only one – obvious way to do it.

Avoid `.length()`, `.getLength()`, `.size()` inconsistencies

Properties

Readability counts, so Python provides a way to avoid writing "getters and setters" when unnecessary.

Properties

Readability counts, so Python provides a way to avoid writing “getters and setters” when unnecessary.
In Java, it’s nearly impossible to make everything public, since changing a class to use getters and setters would require a change of everything that interfaces with it.

Properties

Readability counts, so Python provides a way to avoid writing "getters and setters" when unnecessary.

In Java, it's nearly impossible to make everything public, since changing a class to use getters and setters would require a change of everything that interfaces with it.

Python's properties allow you to make your variable public to begin with, and then write getters and setters only once they are needed to actually check something.

Using Properties

```
class CameraSensor:
    def __init__(self):
        self.brightness = 10

    def take_picture(self):
        # do something
        return image
```

```
camera = CameraSensor()
camera.brightness = 40
camera.take_picture()
```

Using Properties

```
class CameraSensor:
    def __init__(self):
        self._brightness = 10

    def take_picture(self):
        # do something
        return image

    @property
    def brightness(self):
        return self._brightness

    @brightness.setter
    def brightness(self, value):
        if not 0 <= value <= 100:
            raise ValueError
        self._brightness = value
```

```
camera = CameraSensor()
camera.brightness = 40
camera.take_picture()
```