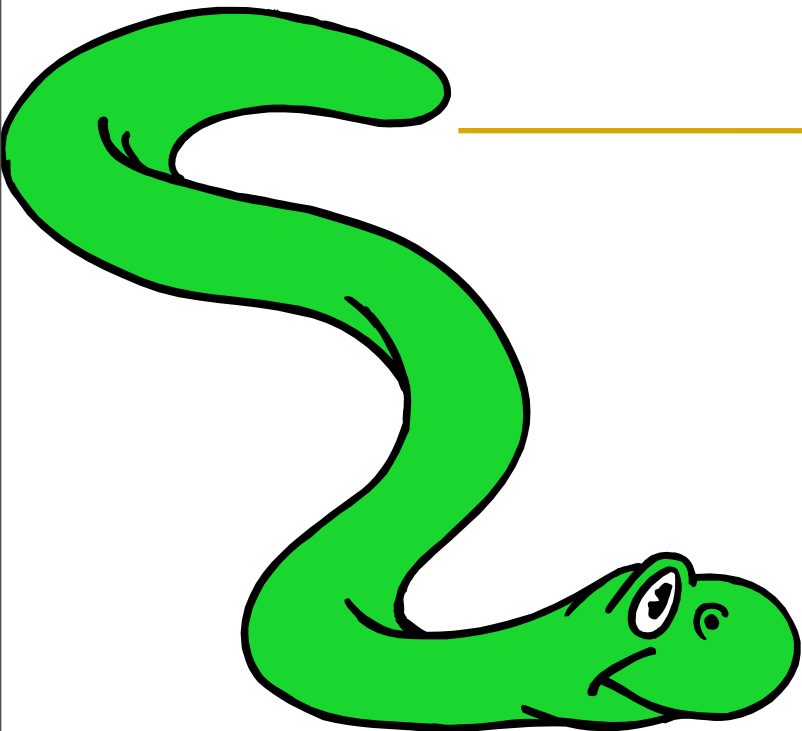

Object Oriented Programming in Python



It's all objects...

- What is object?
 - **data-type**: string, list, dictionary, ...
 - object is defined as **class** in python
- What does object consist of?
 - instance (variable)
 - method (function) -> include instances
 - class -> include both **instances** and **methods**
- Have we seen objects before?

It's all objects...

- Everything in Python is really an object.
 - We've seen hints of this already...
`"hello".upper()`
`list3.append('a')`
`dict2.keys()`
 - You can also design your own objects...
in addition to these built-in data-types.
- In fact, programming in Python is typically done in an object oriented fashion.

Definition of student

- Let us define **student** object
 - class
 - student: represents the student object
 - instance
 - full_name: full name of the student
 - age: age of the student
 - method
 - get_age(): returns the age of the student
 - set_age(num): set student's age to 'num'

Definition of student

Class definition

```
class student:
    def __init__(self, n):
        self.full_name = n
    def get_age(self):
        return self.age
    def set_age(self, num):
        self.age = num
```

__init__ ?
self ?

↑
Indent

Instantiating Objects

- You merely use the class name with () notation and assign the result to a variable.

```
b = student("Jinho Choi")
```

- The arguments you pass to the class name are actually given to its `.__init__()` method.

Constructor: `__init__`

- `__init__` acts like a constructor for your class.
 - When you create a new instance of a class, this method is invoked. Usually does some initialization work.
 - The arguments you list when instantiating an instance of the class are passed along to the `__init__` method.

```
b = student("Jinho Choi")
```

So, the `__init__` method is passed "Jinho Choi".

Constructor: `__init__`

- Your `__init__` method can take any number of arguments.
 - Just like other functions or methods, the arguments can be defined with default values, making them optional to the caller.
- However, the first argument `self` in the definition of `__init__` is special...

Self

- The first argument of every method is a reference to the current instance of the class.
 - By convention, we name this argument `self`.
- In `__init__`, `self` refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called.
 - Similar to the keyword 'this' in Java or C++.
 - But Python uses 'self' more often than Java uses 'this.'

Self

- Although you must specify `self` explicitly when defining the method, you don't include it when calling the method.
- Python passes it for you automatically.

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):  
    self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Traditional Syntax for Access

```
>>> f = student ("Jinho Choi")
>>> f.full_name      # Access an attribute.
"Jinho Choi"
>>> f.age
Error
>>> f.set_age(31)   # Access a method.
>>> f.get_age()
31
>>> f.age
31
```

Accessing unknown members

- What if you don't know the name of the attribute or method of a class that you want to access until run time...
- Is there a way to take a string containing the name of an attribute or method of a class and get a reference to it (so you can use it)?

getattr(object_instance, string)

```
>>> f = student("Jinho Choi")
```

```
>>> getattr(f, "full_name")
```

```
"Jinho Choi"
```

```
>>> getattr(f, "get_age")
```

```
<method get_age of class studentClass at 010B3C2>
```

```
>>> getattr(f, "get_age") ()      # We can call this.
```

```
23
```

```
>>> getattr(f, "get_name")
```

```
Error
```

hasattr(object_instance,string)

```
>>> f = student("Jinho Choi")
```

```
>>> hasattr(f, "full_name")
```

```
True
```

```
>>> hasattr(f, "get_age")
```

```
True
```

```
>>> hasattr(f, "get_name")
```

```
False
```

Two Kinds of Attributes

- The non-method data stored by objects are called attributes. There's two kinds:
 - **Data attribute:**
Variable owned by a particular instance of a class.
Each instance can have its own different value for it.
These are the most common kind of attribute.
 - **Class attributes:**
Owned by the class as a whole.
All instances of the class share the same value for it.
Called "static" variables in some languages.
Good for class-wide constants or for building counter of how many instances of the class have been made.

Two Kinds of Attributes

- Remember assignment is how we create variables in Python; so, assigning to a name creates the attribute.
- Data Attributes
 - Inside the class, you refer to data attributes using `self` - for example, `self.full_name`
- Class Attributes
 - Since there is one of these attributes per class and not one per instance, we use a different notation:
 - We access them using `self.__class__.name` notation.

Two Kinds of Attributes

```
class student:
    def __init__(self, n):
        self.full_name = n

    def set_age(self, num):
        self.dAge = num
        self.__class__.cAge = num

    def prints(self):
        print self.dAge
        print self.__class__.cAge
```

```
>>> a = student('jinho')
>>> a.set_age(31)
>>> a.prints()
31
31
>>> b = student('jeany')
>>> b.set_age(34)
>>> b.prints()
34
34
>>> a.prints()
31
34
```

Data vs. Class Attributes

```
class counter:
    overall_total = 0

    def __init__(self):
        self.my_total = 0

    def increment(self):
        counter.overall_total += 1
        self.my_total += 1
```

```
>>> a = counter()
>>> a.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
1
>>> b = counter()
>>> b.increment()
>>> b.increment()
>>> b.my_total
2
>>> b.__class__.overall_total
3
>>> a.__class__.overall_total
3
```