Classes

Class

A class provides a means to bundle data and functionality together. It creates a new *type* of object

Each instance of class can have

- attributes (data)
- methods (functions)

Namespaces and Scopns

A *namespace* is a mapping from names to objects (also an object)

Examples include

- built-in names like `abs()` and `Exception`
- global module names like `math` and `sys`
- local names like `x` and `y` in a function

The *important* thing to know is that two namespaces is that there is **no** relation between names in different namespaces

For example, `math.pi` and `sys.pi` are two different objects

Namespaces are created at different moments and have different lifetimes

Scope

is a region of python code where a namespace is directly accessible.

When you refer to a name, python searches:

1. the inner most scope, local names
2. the scope of any enclosing functions, starting at the nearest, nonlocal names
3. current modules global names
4. built-in names

Example

```python
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Output:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Classes

Classes

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Very similar to a function

Usually the insides of a class is functions, but other statements are allowed and is often useful

When a new class is created, a new namespace is created, and is uesd as the local scope

Class objects

```
class Myclass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

My class objects have two kinds of operations, attribute references and instantiation

References have a dot ntation `obj.name` and returns a value

`Myclass.i` returns the value `12345` and `Myclass.f` returns a function object

Instantiation

Instantiation is done by calling the class object, and it returns a new instance of the class

```
x = MyClass()
```

This creates an empty object, but sometimes we want to initialize it with some values

We can use the special method `__init__()`, which is run when the class is instantiated

```
def __init__(self):
    self.data = []
```

Instantiation

And the `__init__()` method can also have arguments which are given when the class is instantiated

```python
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
```

Instance Objects

The only opertions understood by instance objects are attribute references

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

note that `x.counter` will create a new attribute if it does not already exist, like how regular variables work

Method objects

A method (a function defined in a class) is usually called after it is bound

```
x.f()
```

In the `MyClass` example, this will return `'hello world'`, but you don't have to call the method right away

```
xf = x.f
while True:
    print(xf())
```

Note that this doesn't have an argument even though `f()` is defined with one

the first argument `self` is a special argument that is equivalent to the instance object itself

Example

```
class Dog:
    kind = "canine"

    def __init__(self, name):
        self.name = name
```

If I call

```
d = Dog("Fido")
e = Dog("Buddy")
```

Output

```
d.kind
>>> 'canine'

e.kind
>>> 'canine'

d.name
>>> 'Fido'

e.name
>>> 'Buddy'
```

Shared data can have possible surprising effects

Example

```python
class Dog:
    tricks = []

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)
```

Output

```
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over', 'play dead']
```

if you want each do to have their own tricks, you should define `tricks` in the `__init__()` method

Example

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

Summary

- Classes are a way to bundle data and functionality together
- Classes create a new type of object
- Classes have attributes (data) and methods (functions)
- Classes create a new namespace
- Classes can have an `__init__()` method to initialize the object