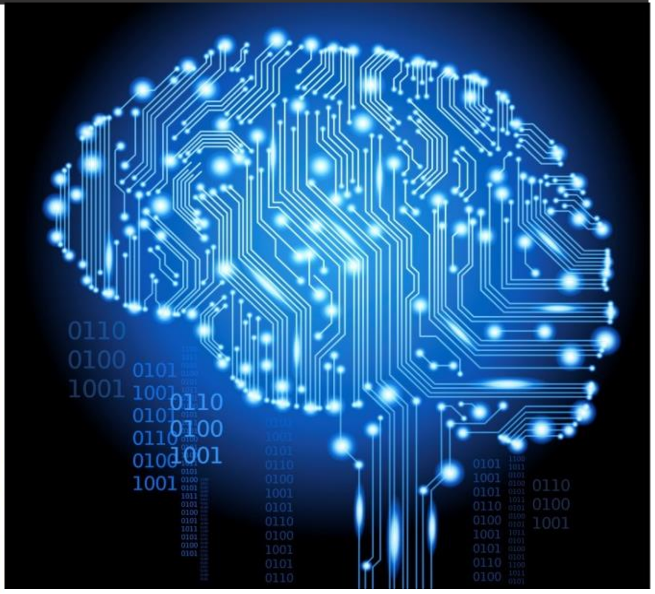
MONASH University

Information Technology

FIT1008 & 2085 Lecture 15

Classes & Objects

Prepared by:
Maria Garcia de la Banda, Pierre Le Bodic



Where are we at…

- Basic exception handling, assertions and unit testing
- How to implement ADTs for list

Objectives for today’s lecture

- **To learn a little bit about Object Oriented Programming in Python**
- **In particular, to learn:**
  - How to define basic classes
  - How to instantiate them into objects
  - How to define methods and how to use them
  - An example of classes for implement lists
  - The importance of namespaces and scoping rules
- **We will NOT learn about a major OO component: inheritance**
  - Not needed for this unit
  - Even though it is central to the idea of OO
  - Note: Python and Js inheritance is quite different

Objects

- **We have already seen objects**
  - But only as blocks of memory containing some data
- **As you know, objects are more than data fields (or data attributes):**
  - They have methods that can be performed by/to the object
- **They are similar to real life "objects"**
- **Example: a human "object" would have:**
  - Data describing the human (by its attributes):
    - Name, age, gender, height, weight, eye colour, race, etc
  - Methods that can be performed by/on the human:
    - Eat, sleep, run, study, die, etc
  - Both form the attributes of the object
  - Python's attributes is what you used to call "properties" in Js
- **Remember, in Python every value is an object**
  - In Js there were "primitive types". Not in Python.

Some of these data attributes change with time, others don't

Some change the state of the human's data attributes, some don't

Using Objects

- **We say that:**
  - Every value is an object
  - They have both data and methods
- **So, how do we access the data and methods of an object?**
  - Through the “dot” notation:

Same as Js

```
>>> "abcd".upper()
'ABCD'
>>> x = [1,2,3,4]
>>> x.append(5)
>>> x
[1, 2, 3, 4, 5]
```

- **Also referred to as “qualifying” a variable or method**
  - We will see later why
- **The dot notation is common to many languages**

Classes

In Python you cannot create objects without a class!

- **A class is a blueprint for objects**
- **That is, a class defines its attributes:**
  - The data stored by the object
  - The methods that can be used by the object
  - How it is constructed

So, kind of combination of constructor and prototype in JS

Class syntax now also in Js for prototypes

- **Classes are defined using the following syntax:**

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

class is a keyword

The body is indented

Class header

By convention, the class name starts with an upper case

Class body

where the most common statement is a method definition

- **Every object is created by instantiating a class (see later)**
  - That is why we say that an object is an instance of a class

Methods

- **Define operations that can be performed by any object created as an instance of the class**
- **A method definition looks like a function definition**
- **Except that:**
  - It must appear inside the class
  - Its first argument must be a reference to the instance whose method was called
    - By (a very strong) convention, this argument is named **`self`**
    - You must explicitly add **`self`** to the method definition
    - But it is automatically added in a method call

The `__init__` method

The constructor in JS

- **By convention, (if defined) it is the first method in a class**
  - First code executed when creating an instance (automatically!)
- **As in all other methods, its first argument is a reference to the instance whose method was called**

Don't forget **`self`**

- **Consider for example the `Point` class:**
  - It has two instance variables:
    - **`self.xCoord`**
    - **`self.yCoord`**
  - They start with **`self.`** and thus, they belong to the instance:
    - Their value is specific to each instance
  - But their name is global to all methods in the class:
    - Means: you can access it from every method in the class
  - If a variable does not start with **`self.`** it is local to the method that binds it (**`x`** and **`y`**)

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

MONASH University

How do we instantiate a class?

- **Easy: call it as if it were a function with the `__init__` arguments**

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

Let's put this code in file **`point.py`**

No need to call `__init__` Done automatically when creating the instance

Every class instance has some built-in attributes like `__class__`

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.xCoord
1
>>> p1.yCoord
3
>>> p2 = point.Point(-4,7)
>>> p2.xCoord
-4
>>> p2.yCoord
7
>>> p1.__class__
<class 'point.Point'>
```

Why is `point.` needed? We'll see

`p1.` notation lets us access the object's attributes

MONASH University

Reconsidering the List ADT

- **If we had known about classes we could have implemented it as:**

```
class List:
    def __init__(self,size):
        self.the_array = [None]*size

    def length(self):
        return len(self.the_array)

    def is_empty(self):
        return self.length() == 0

    def lin_search(self, item):
        for element in self.the_array:
            if item == element:
                return True
            return False
```

Note that every method has `self` as first argument

From the user's perspective, the use of the ADT is identical, except for (a) not having to pass the list as an argument and (b) using the dot notation to use the methods (or any other attribute)

Your turn: add **insert**, **pop**, **remove**, **copy**, **clear**

Before and after for the List ADT

- Has the usage changed that much?
- Not really:

Before	After
`l= List(10)`	`l= List(10)`
`append(l, 1)`	`l.append(1)`
`append(l, 2)`	`l.append(2)`
`finished = is_empty(l)`	`finished = l.is_empty()`
`found = lin_search(l, item)`	`found = l.lin_search(item)`
`item = pop(l, i)`	`item = l.pop(i)`
`clear(l)`	`l.clear()`

Recap

- **We create a class:**
  - By simply writing `class Name:` in some file
  - And adding indented statements (such as methods) to it
    - All methods have `self` as first argument
- **We create an object:**
  - By instantiating the class
  - That is, by simply calling `Name` with the appropriate arguments
    - Those used by method `__init__`
  - The object has access to all the attributes defined by the class using the "dot" notation (e.g., `the_list.append`)

And if the class does not have __init__?

- **The `__init__` method is not mandatory**
- **One could easily define:**

```
>>> class Silly:
...       i = 8
...
>>> Silly.i
8
>>> s1 = Silly()
>>> s1.i
8
>>> s2 = Silly()
>>> s2.i
8
```

A class variable. Defined in a class but outside any method. Its value is shared by all instances of the class

Belongs to the class. Thus, it exists without object and is accessed through the class

Can also be accessed through an instance (see later)

All instances of the class have the same value for **i**

```
>>> Silly.i = 11
>>> s1.i
11
>>> s2.i
11
>>> s1.i = 6
>>> s1.i
6
>>> s2.i
11
```

To modify **i** you must do it through the class not through an instance!

What? if **i** is a class variable, shouldn't it be 6?

MONASH University

Names

- **First remember, in Python all identifiers are names:**
  - Variables, functions, methods, modules, types, …
  - This means, a name can only refer to one thing at a time!
- **Careful when reusing names then…**

```
>>> a_name = 10*6
>>> a_name
60
>>> def a_name(x):
...     return x*100
...
>>> a_name
<function a_name at 0x100520560>
>>>
```

```
>>> a_name = 'hello'
>>> a_name
'hello'
>>> class a_name:
...     i = 8
...
>>> a_name
<class '__main__.a_name'>
>>>
```

All the above code created a single variable named **`a_name`**

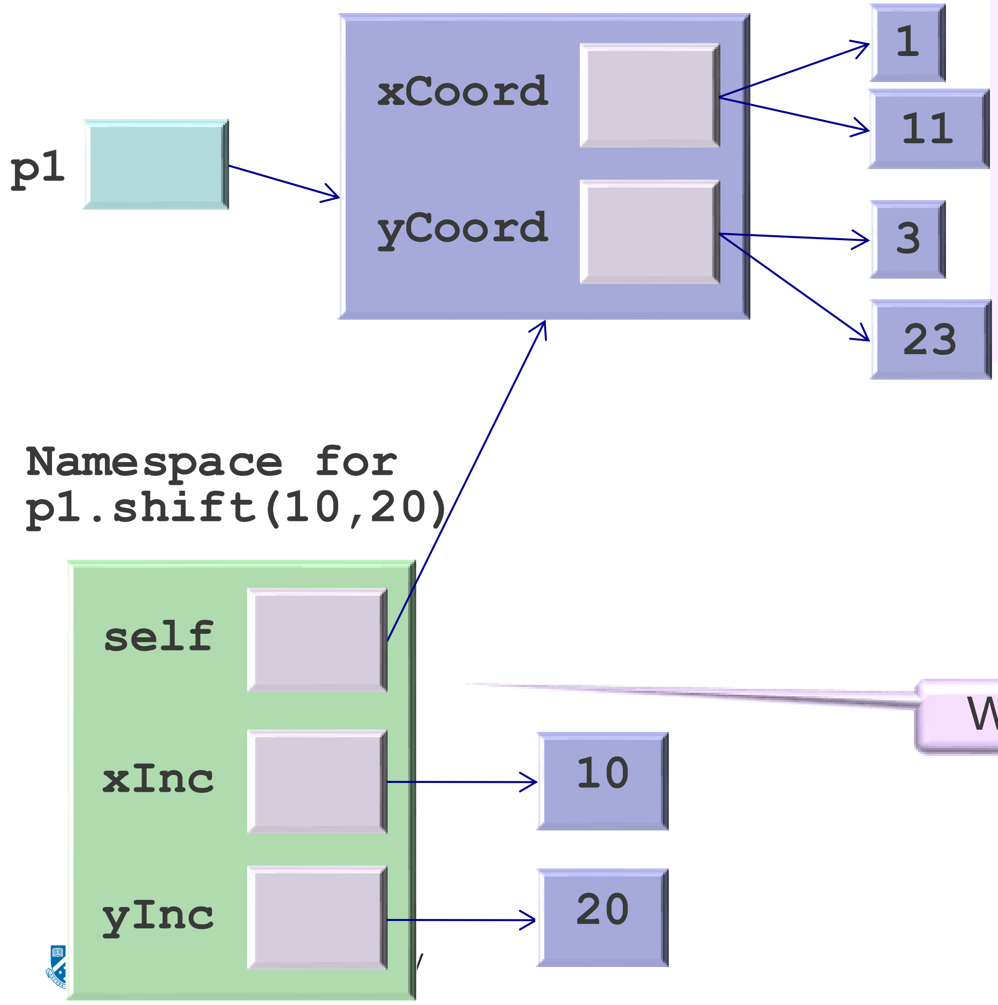
Namespaces (or environments)

- **A namespace is a mapping of names to objects**
  - Like a dictionary
- **For example, when the interpreter starts:**
  - It creates a namespace with the names of the built-in functions
- **Each file (also called module) has its own namespace**
  - Don’t put two classes or two functions with the same name in a file
  - They share the same namespace, so the result can be surprising:
    - With two functions, the second definition overwrites the first
- **Functions have their namespace too. When a function is called:**
  - Python creates a local namespace for it
  - That namespace is forgotten once the function finishes
- **Names belong to the namespace in which they are bound**

Like a stack frame in MIPS

Namespaces (cont)

- An example with the `Point` class



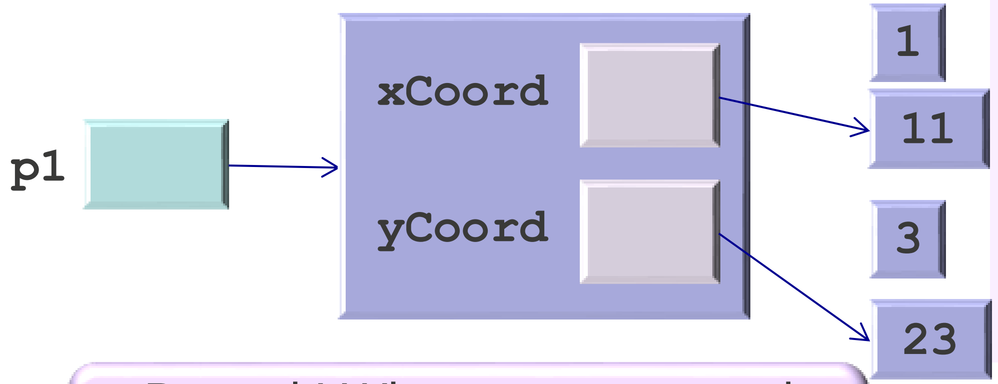
```
class Point:
    def __init__(self,x,y):
        self.xCoord = x
        self.yCoord = y

    def shift(self, xInc, yInc):
        self.xCoord += xInc
        self.yCoord += yInc
```

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.shift(10,20)
```

While `p1.shift(10,20)` is executing

Namespaces (cont)

- An example with the `Point` class



But wait! Why were **`xInc`** and **`yInc`** in **`shift`**'s namescape? Were they "bound" by **`shift`**?

When **`p1.shift(10,20)`** finishes executing the namespace is forgotten

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

    def shift(self, xInc, yInc):
        self.xCoord += xInc
        self.yCoord += yInc
```

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.shift(10,20)
>>> p1.xCoord
11
>>> p1.yCoord
23
>>>
```

MONASH University

Binding a name

- **There are many ways to bind a name in Python**
- **For example, by:**
  - Assigning to a variable (`x = 13`)
  - Receiving an argument (that's the case for `xInc` and `yInc`)
  - Importing a module (`import x`)
  - Importing a variable (`from y import x`)
  - Defining a function (`def x(foo): ...`)
  - Defining a class (`class x: ...`)
  - Writing a for loop (`for x in y: ...`)
  - Writing an except clause (`try: ... except x: ...`)
- **If any of these appears inside a function:**
  - It makes the name local to the function

Scoping

- **Scope: block of text where a namespace is directly accessible**
  - That is, where there is no need to "qualify" the name
    - `from silly import Silly` brings `Silly` to the current namespace
    - Then, there is no need for `silly.Silly`
- **Often there are several scopes in operation, for example:**
  - The scope of the method that is executing
  - The scope of the class where the method is defined
  - The scope of the module where the class is defined
  - The scope of the interpreter that is executing
  - etc
- **Scope is determined statically but used dynamically**
  - Statically: so that you can always determine the scope of any name by looking at the program
  - Dynamically: that it, is at run-time that Python searches for names

Scoping Rules

- **Remember: names belong to the namespace where they are bound**
  - The scope of a name does not change while the program is running
- **During execution, Python searches for names as follows:**
  - First, in the innermost scope
    - Contains all the local names (those in the method's namespace)
  - Then, in the scopes of any enclosing functions:
    - Searched from the nearest-to-outer enclosing scope
    - Contains nonlocal and nonglobal names
  - Then the current module's global names
    - That is, those in the module's namespace
  - Last, the namespace containing built-in names
- **Programmers can change the scope of identifiers**
  - But we are not going to see this

Example of Scoping Rules

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
>>> a()
a
>>> c(7)
7
>>> d()
second
>>> e()
e
```

a() has **x** in its local namespace with value **a**

b() does not have **x** in its local namespace. And it is not defined within a function. So it looks at the enclosing namespace (module/interpreter) where it finds **x** with value **'first'**

b() as before but the value of the global **x** is now **'second'**

a() as before

c() has **x** in its local namespace with value **7**

d() calls **b()** which is as before (the value of the global **x** is still **'second'**)

e() defines and calls **f()** which does not have **x** in its local namespace, so it looks in that of its enclosing function and finds it with value **'e'**

MONASH University

How does this relate to “qualifying”?

- **Remember the code used some slides before:**

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

We said “why is `point.` needed?

Because the name `Point` is not directly accessible from the current code, i.e., not in its namespace or in any one where Python will search for it

Qualifying it by `point.` allows us to access the namespace of module `point.` which contains the name `Point`

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.xCoord
1
>>> p1.yCoord
3
>>> p2 = point.Point(-4,7)
>>> p2.xCoord
-4
>>> p2.yCoord
7
>>> p1.__class__
<class 'point.Point'>
```

And how does it relate to this case?

- **Remember the case a few slides before, where:**

```
>>> class Silly
...       i = 8
...
>>> Silly.i
8
>>> s1 = Silly()
>>> s1.i
8
>>> s2 = Silly()
>>> s2.i
8
>>> Silly.i = 11
```

```
>>> s1.i
11
>>> s2.i
11
>>> s1.i = 6
>>> s1.i
6
>>> s2.i
11
>>> Silly.i = 22
>>> s1.i
6
>>> s2.i
22
```

We said: what? if **i** is a class variable, shouldn't it be 6?

No, Because this assignment created a new attribute for **s1** (but not for **s2**) in its local namespace

This will first look in **s1**'s local namespace and will find the newly created attribute

This will first look in **s2**'s local namespace, will find nothing and then look at the class namespace where class variable **i** exists

Scoping rules: A few more examples

- **What would this print?**

```
x = 77

class Mine:
    x = 6
    y = 10
    z = 20
    def __init__(self,x):
        self.y = 12
        self.x = 5
```

```
>>> a = Mine(0)
>>> a
<__main__.Mine object at 0x1006bda10>
>>> a.x
5
>>> a.y
12
>>> a.z
20
>>> b = Mine(0)
>>> a.z = 8
>>> a.h = 9
>>> b.z
20
>>> Mine.x
6
>>> b.h
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Mine' object has no
attribute 'h'
```

Instance variable (a.x)

Instance variable (a.y)

Class variable (a.z)

Creates new instance variables in **a** (a.z = 8, a.h = 9)

Class variable (b.z)

Class variable (Mine.x)

Non-existent variable in **b** (b.h)

MONASH University

Scoping rules: A few more examples

- **What would this print?**

```
x = 67

def one(y):
    if y:
       print(x)
    def x(a):
        return a * 2
    print(x)

def two():
    for x in range(3):
        print(x)

def three():
    class x:
        x = 3
        print(x)
    print(x)
```

```
>>> one(False)
<function one.<locals>.x at 0x1006bf440>
>>> one(True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in one
UnboundLocalError: local variable 'x'
referenced before assignment
>>> two()
0
1
2
>>> three()
3
<class '__main__.three.<locals>.x'>
```

Non-bound local variable

local variable

MONASH University

Scoping rules: A few more examples

- **What would the following print?**

```
x = [4]

def a():
    x.append(3)

def b():
    x = [1, 2]
    x.append(3)
    print(x)

def c(x):
    for x in range(3):
        print(x)

def d():
    print(x)
    x = 7
    print(x)
```

```
>>> a()
>>> x                          Global variable
[4,3]
>>> b()                        Local variable
[1,2,3]
>>> c(100)                     Local variable
0
1
2
>>> d()                        Non-bound local variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in d
UnboundLocalError: local variable 'x'
referenced before assignment
```

MONASH University

Summary

- **Classes and instances (objects)**
- **Methods (the __init__ method in particular)**
- **Names, Namespaces, Scopes and Scoping Rules**