UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FACULTY OF
COMPUTER SCIENCE

Object Oriented Programming

Paradigma prosedural

```
#Prosedural Paradigm
print "Hello World!"
print "Hello Again"
print "I like typing this."
print "This is fun."
print 'Yay! Printing.'
print "I'd much rather you 'not'."
print 'I "said" do not touch this.'
```

Object

- Lihat sekeliling Anda !
- Anda akan menemukan banyak sekali objek:
  - Meja
  - Papan tulis
  - Komputer
  - Gedung
  - dst

Object

- Ketika Anda melihat teman Anda, Anda melihat teman Anda tersebut sebagai sebuah **Objek**.

- Objek adalah sebuah entitas yang mempunyai **data/atribut** serta ***behavior***.

Object

Sebuah objek mempunyai:

- Attribute/instance variable/property
  - "Sesuatu yang objek **tahu**"
  - **Data** dari sebuah objek
  - properti/atribut dari objek
- Method/behavior
  - "sesuatu yang objek **lakukan**"
  - Memanipulasi **data** dari objek yang bersangkutan

Object

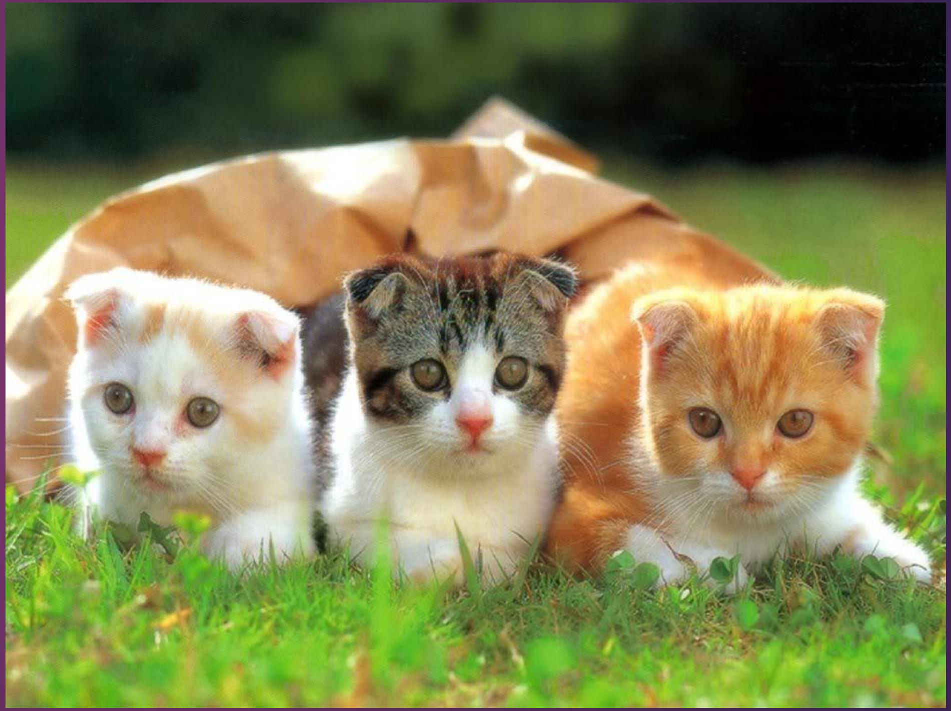
- Sebuah objek = **Kucing**

attribute/variable/property:

- nama
- ukuran
- warna

Method/behavior

- mengeong()
- makan()
- tidur()
- mengejarTikus()



Object

- Sebuah objek = **Movie**

attribute/variable/property

- judul
- genre
- rating

Method/behavior

- playIt()
- setJudul()
- getJudul()

Latihan

- Lihatlah ke sekeliling Anda !
- Cari sebuah objek
- Sebutkan atribut/properti dari objek tersebut !
- Sebutkan method/behavior yang mungkin untuk objek tersebut !

Class

- *Blueprint/Plan/Template* untuk sebuah objek.

- Misal, seorang arsitek yang ingin membangun sebuah gedung:
  - Gambar gedung di kertas : kelas
  - Gedung yang sudah dibangun : objek/instansiasi dari kelas

Class: where object belongs to

- Objek yang jenisnya sama ➔ berada pada satu **class** yang sama.
- Class: mendefinisikan implementasi detil dari objek
  - Semua methods (fungsi yang hidup dalam sebuah class) yang dapat digunakan
  - Instruksi/kode dari methods tersebut ➔ perilaku/behaviour
  - keadaan/state/attribut dari object yang disimpan
  - Berdasarkan definisi class ➔ object dapat dibuat.

Class

- Kelas digunakan sebagai basis untuk menentukan bagaimana membangun (menghidupkan) sebuah objek.
- Proses menghidupkan sebuah objek dari kelas = **instansiasi** (*instantiation*).
- Objek tidak bisa diinstansiasi tanpa sebuah kelas.

Objek adalah instansiasi dari Kelas

Kelas

Dog
Name Size Color

Bark()
ChaseACat()

Objek 1 : black

Objek 2 : brown

Objek 3 : white

Representasi dan Abstraksi

- Kita tidak dapat menyimpan benda atau obyek dari dunia nyata dalam komputer.
- Komputer menyimpan ciri-ciri penting dari benda sesuai dengan tujuan/keperluan kita.

Kelas vs Objek

Kelas

- Deskripsi atribut dari sekumpulan objek
- Sebuah konsep
- Kelas dibuat ketika kita *coding*, atau menuliskan kode sumber kita
- Kelas ada di program/kode sumber kita
- Contoh 1: person

Objek

- Representasi dari sebuah instansiasi
- Sebuah fenomena
- Objek akan ada setelah program kita dieksekusi
- Objek hidup/tinggal di memori komputer
- Contoh 1: Soekarno, Soeharto, Habibie

Class Definition

- Class definitions have the form

```
class <class-name> (<superclass>, ...):
<variable and method definitions>
```

- Methods look a lot like functions! Placing the function inside a class makes it a method of the class, rather than a standalone function.
- The first parameter of a method is usually named `self`, which is a reference to the object on which the method is acting.

Example

```
# Circle.py
import math
class Circle(object):


    def __init__(self, radius = 1):
        self.radius = radius


    def __str__(self):
        return "Circle with radius {}".format(self.radius)


    def getPerimeter(self):
        return 2 * self.radius * math.pi


    def getArea(self):
        return math.pi * (self.radius ** 2)


    def setRadius(self, radius):
        self.radius = radius
```

Example

```
>>> myCircle = Circle()
>>> print(myCircle)
Circle with radius 1
>>> myCircle.getPerimeter()
6.283185307179586
>>> myCircle.getArea()
3.141592653589793
>>> myCircle.setRadius(5)
>>> print(myCircle)
Circle with radius 5
```

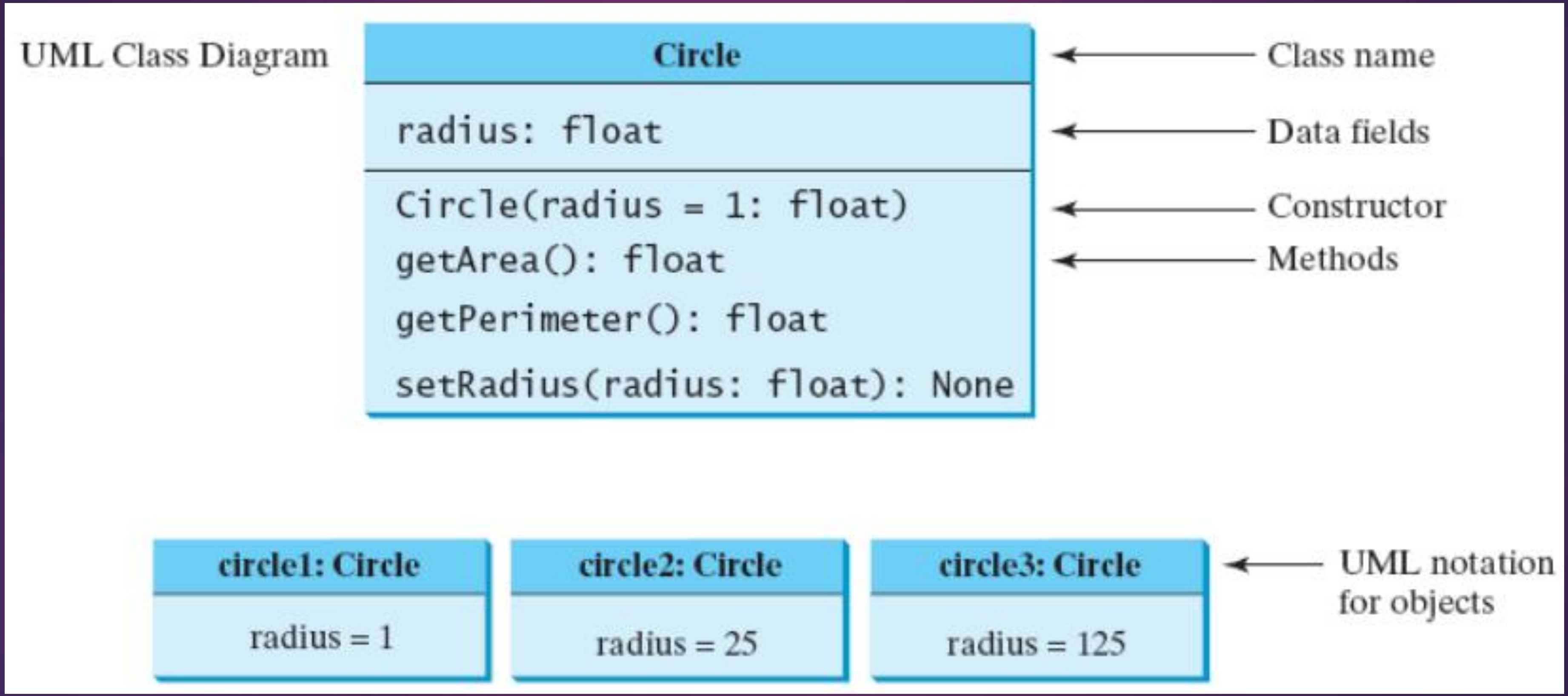
Example

```
# TestCircle.py
from Circle import Circle


def main():
    # Create a circle with radius 1
    circle1 = Circle()
    print("The area of radius {} is {:.2f}.".format(circle1.radius, circle1.getArea()) )
    # Create a circle with radius 25
    circle2 = Circle(25)
    print("The area of radius {} is {:.2f}.".format(circle2.radius, circle2.getArea()) )
    # Modify circle radius
    circle2.setRadius(100)
    print("The area of radius {} is {:.2f}.".format(circle2.radius, circle2.getArea()) )
main()
```

```
>>>
The area of radius 1 is 3.14.
The area of radius 25 is 1963.50.
The area of radius 100 is 31415.93
```

Class Diagram



UML Class Diagram

Circle

radius: float

Circle(radius = 1: float)
getArea(): float
getPerimeter(): float
setRadius(radius: float): None

Class name

Data fields

Constructor

Methods

circle1: Circle
radius = 1

circle2: Circle
radius = 25

circle3: Circle
radius = 125

UML notation for objects

Everything in Python is Object

- Python embraces OOP at fundamental level
- OOP Principles:
  - ***encapsulation***: The class implementation details are invisible (hidden) from the user. All interaction with an object occurs through a well-defined interface that supports a modular design.
  - ***inheritance***: The ability to derive a new class from one or more existing classes. Inherited variables and methods of the original (parent) class are available in the new (child) class as if they were declared locally.
  - ***polymorphism***: An object-oriented technique by which a reference that is used to invoke a method can result in different methods being invoked at different times, based on the type of the actual object referred.

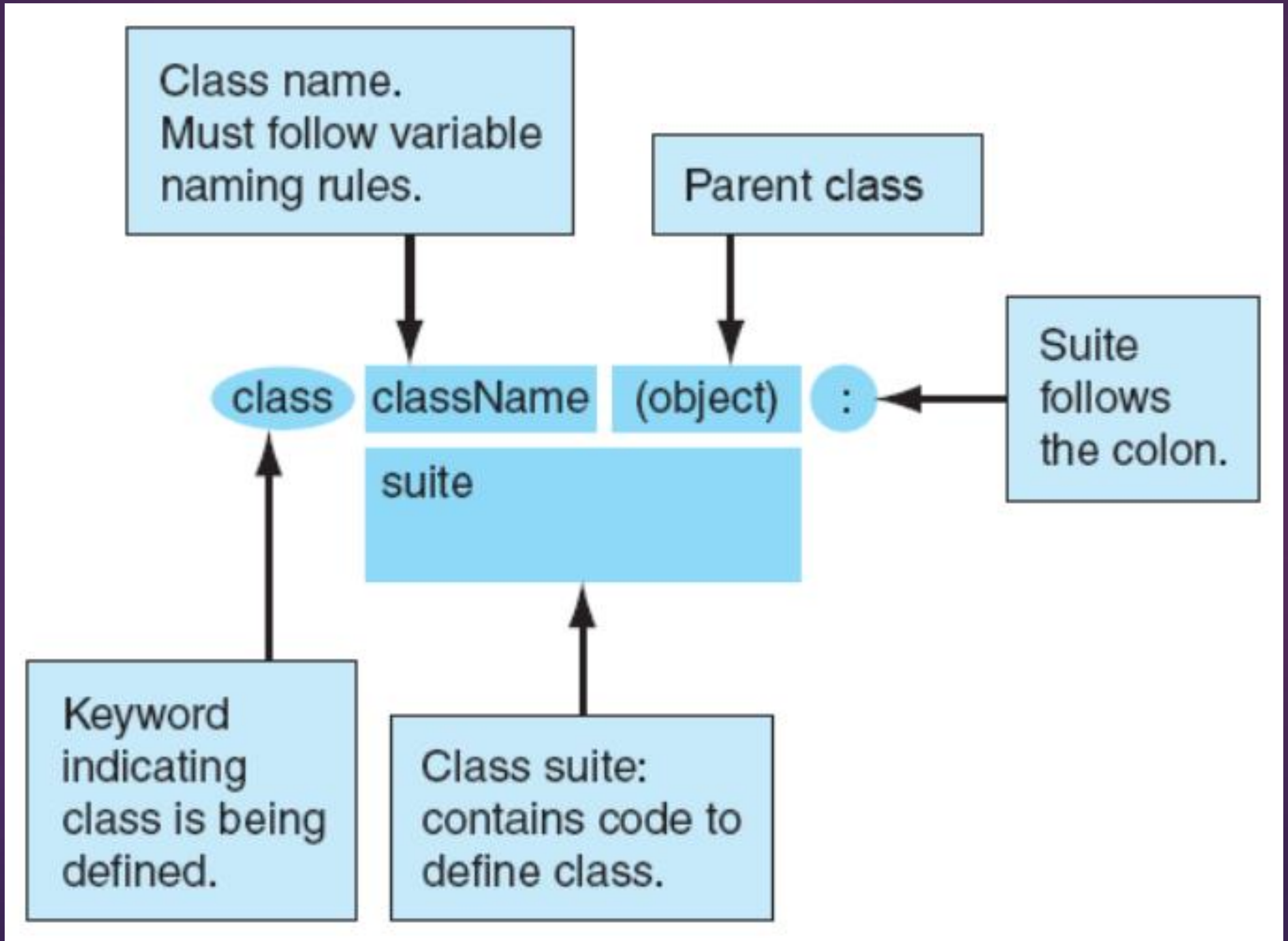
Why a class

- We make classes because we need more complicated, user-defined data types to construct instances we can use
- Each class has potentially two aspects:
  - The data(type, number, names) that each instance might contain
  - The messages/methods that each instance can respond to

Standar Class Name

- The standard way to name a class in Python is called ***CapWords***:
  - Each word of a class begins with a Capital letter
  - no underscores
  - sometimes called ***CamelCase***
  - makes recognizing a class easier

Basic format for Class


Class name.
Must follow variable naming rules.
Parent class
class
className
(object)
:
Suite follows the colon.
suite
Keyword indicating class is being defined.
Class suite: contains code to define class.

dot reference

- we can refer to the attributes of an object by doing a dot reference, of the form: `object.attribute`
- The attribute can be a variable or a method
- It is part of the object
- Example
  - `print(my_instance.my_val)`
    - print a variable associated with the object `my_instance`
  - `my_instance.my_method()`
    - call a method associated with the object `my_instance`
  - variable versus method:

    you can tell by the parentheses at the end of the expression

Method

Method vs Function

- As discussed before, a method and a function are closely related.
- They are both “small programs” that have parameters, perform some operation and return a value
- main difference is that methods are functions tied to a particular object

Difference in Calling

- Methods are called in the context of an object:
  - function:
    `do_something(param1)`
  - method:
    `an_object.do_something(param1)`
- This means that the object that the method is called on is *always implicitly a parameter*!

Difference in definition

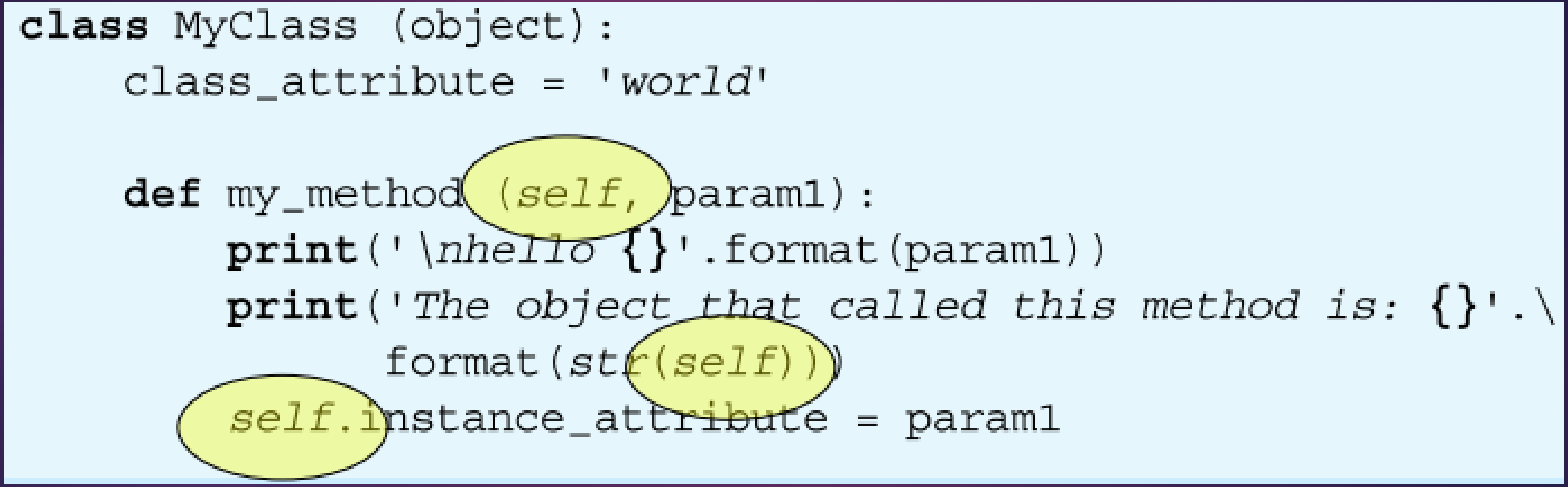
- Methods are defined ***inside*** the body of a class
- Methods always bind the first parameter in the definition to the object that called it
- This parameter can be named anything, but traditionally it is named ***self***

```
class MyClass(object):
def my_method(self,param1):
...
```

`self`

- `self` is an important variable. In any method it is bound to the object that called the method
- Through `self` we can access the instance that called the method (and all of its attributes as a result)

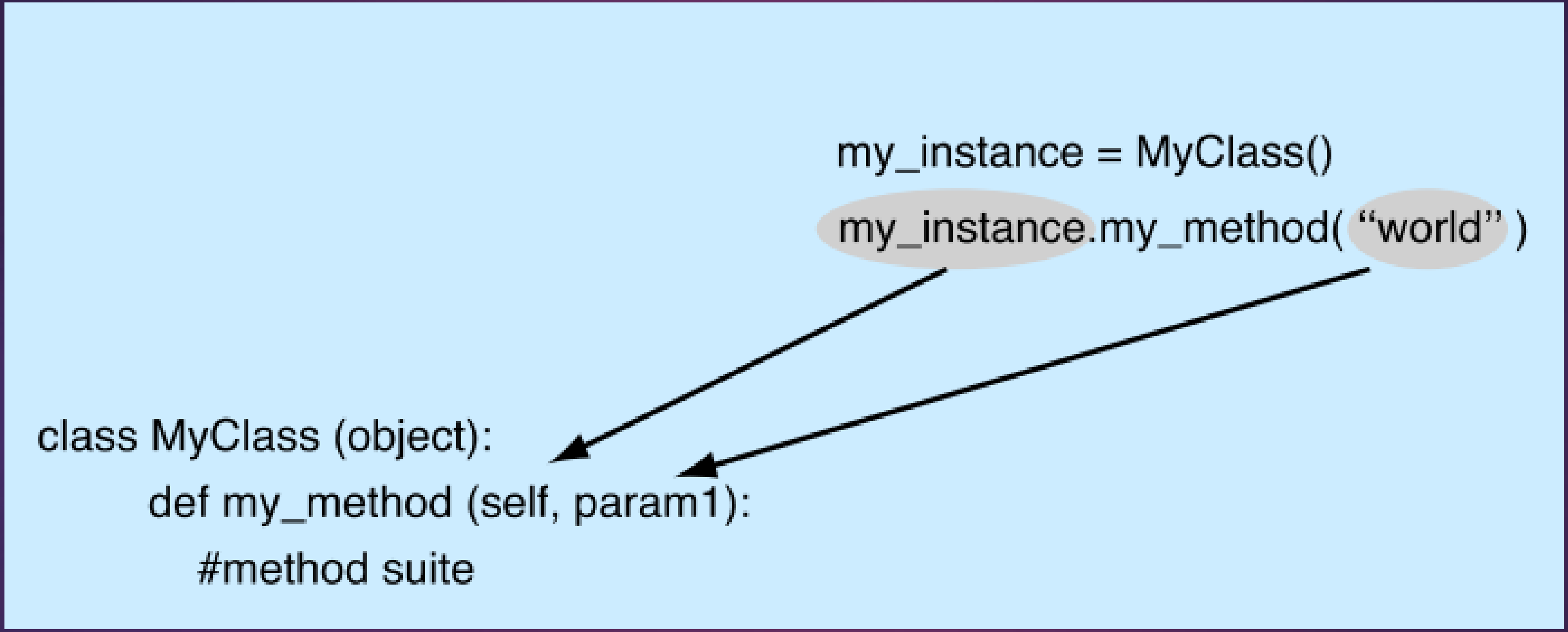
Example



```
class MyClass (object):
    class_attribute = 'world'

    def my_method (self, param1):
        print('\nhello {}'.format(param1))
        print('The object that called this method is: {}'.\
              format(str(self)))
        self.instance_attribute = param1
```

Binding `self`

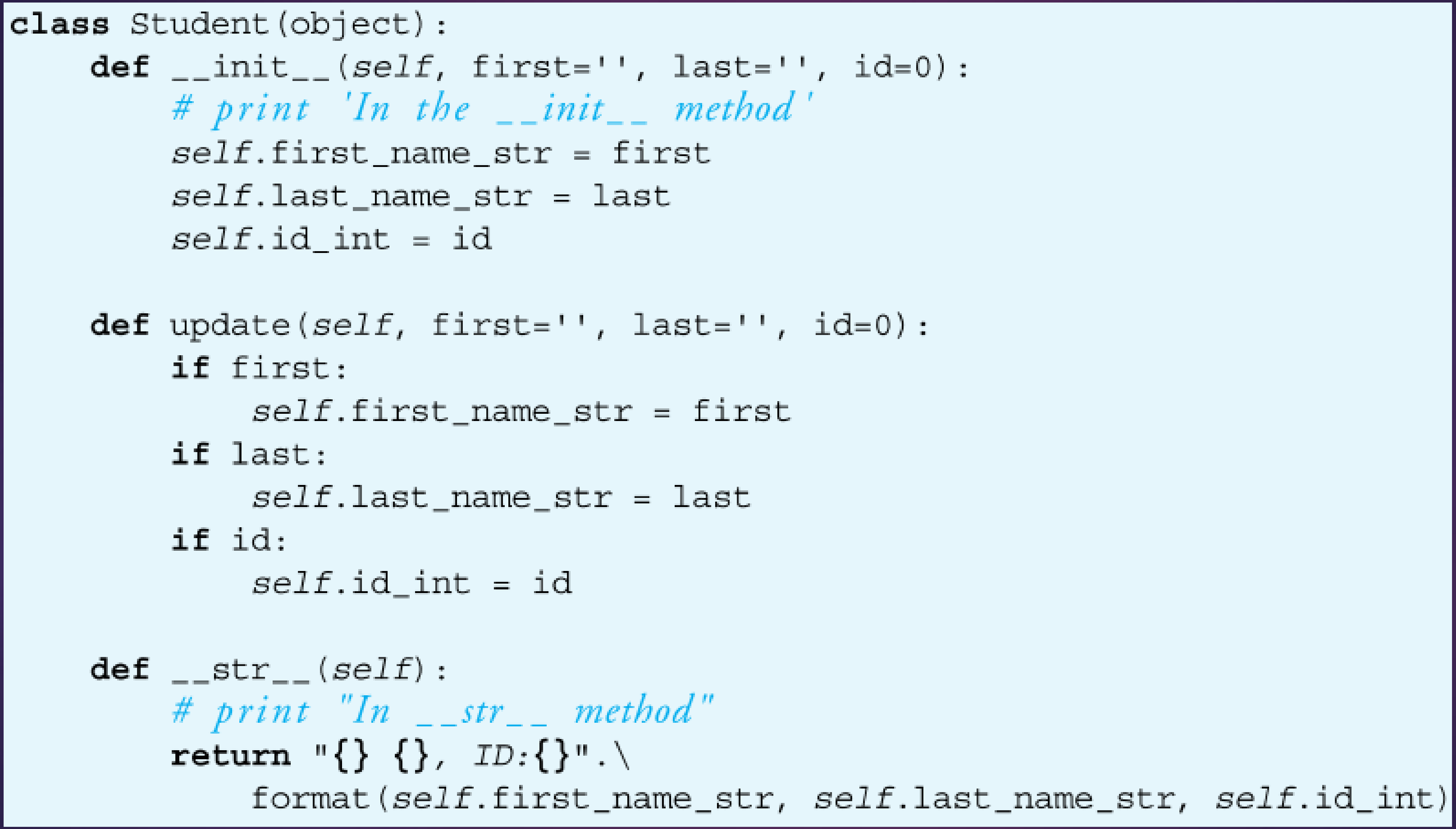


```
my_instance = MyClass()
my_instance.my_method( “world” )
```

```
class MyClass (object):
        def my_method (self, param1):
            #method suite
```

`self` bound automatically

- when a method call is made, the object that called the method is **automatically** assigned to `self`
- we can use `self` to remember, and therefore refer, to the calling object
- to reference any part of the calling object, we must always precede it with `self.`



```
class Student(object):
    def __init__(self, first='', last='', id=0):
        # print 'In the __init__ method'
        self.first_name_str = first
        self.last_name_str = last
        self.id_int = id

    def update(self, first='', last='', id=0):
        if first:
            self.first_name_str = first
        if last:
            self.last_name_str = last
        if id:
            self.id_int = id

    def __str__(self):
        # print "In __str__ method"
        return "{} {}, ID:{}".\
            format(self.first_name_str, self.last_name_str, self.id_int)
```

Python Standard Method

- Python provides a number of standard methods which, if the class designer provides, can be used in a normal "Python“ way
  - many of these have the double underscores in front and in back of their name
  - example: `__str__`

Standar Method : Constructor

- The constructor method is called when an instance is made
- The constructor method sets up the instance with variables, by assignment
- As mentioned, a constructor is called by using the name of the class as a function call (by adding () after the class name)

```
student_inst = Student()
```

  - creates a new instance using the constructor from class Student

Standar Method : Constructor

- one of the special method names in a class is the constructor name, __init__
- by assigning values in the constructor, every instance will start out with the same variables
- you can also pass arguments to a constructor through its init method

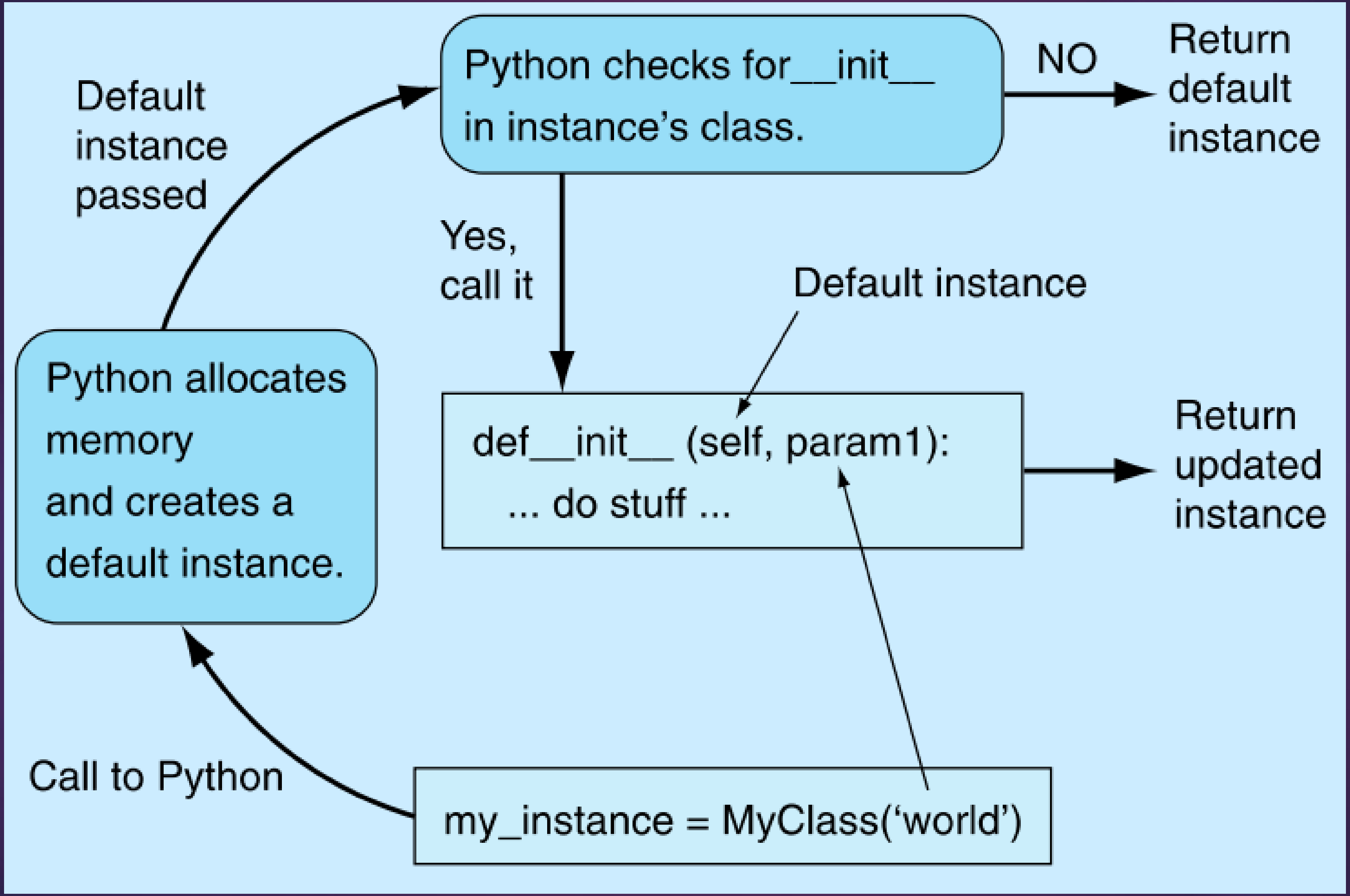
Example : Student Constructor

```
def __init__(self, first='', last='', id=0):
self.first_name_str = first
self.last_name_str = last
self.id_int = id
```

- `self` is bound to the default instance as it is being made
- If we want to add an attribute to that instance, we modify the attribute associated with `self`

Example:

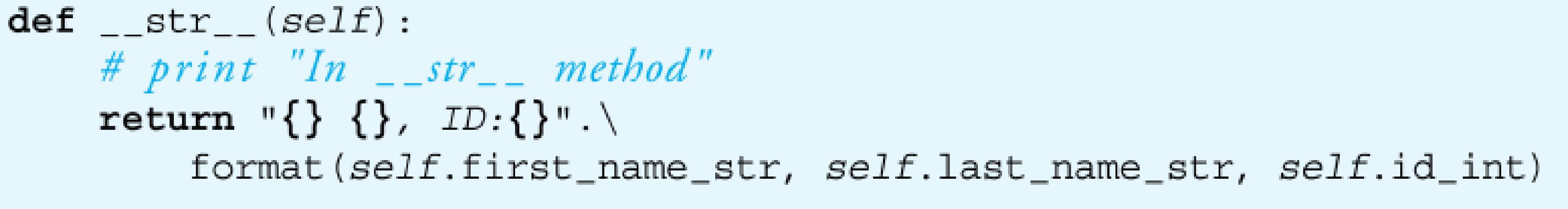
```
>>> s1 = Student()
>>> print(s1.last_name_str)
>>> s2 = Student(last='Python',
first='Monty')
>>> print(s2.last_name_str)
Python
>>>
```


Python checks for__init__ in instance's class.
NO
Return default instance
Default instance passed
Yes, call it
Default instance
Python allocates memory and creates a default instance.
def__init__ (self, param1):
... do stuff ...
Return updated instance
Call to Python
my_instance = MyClass('world')

Default Constructor

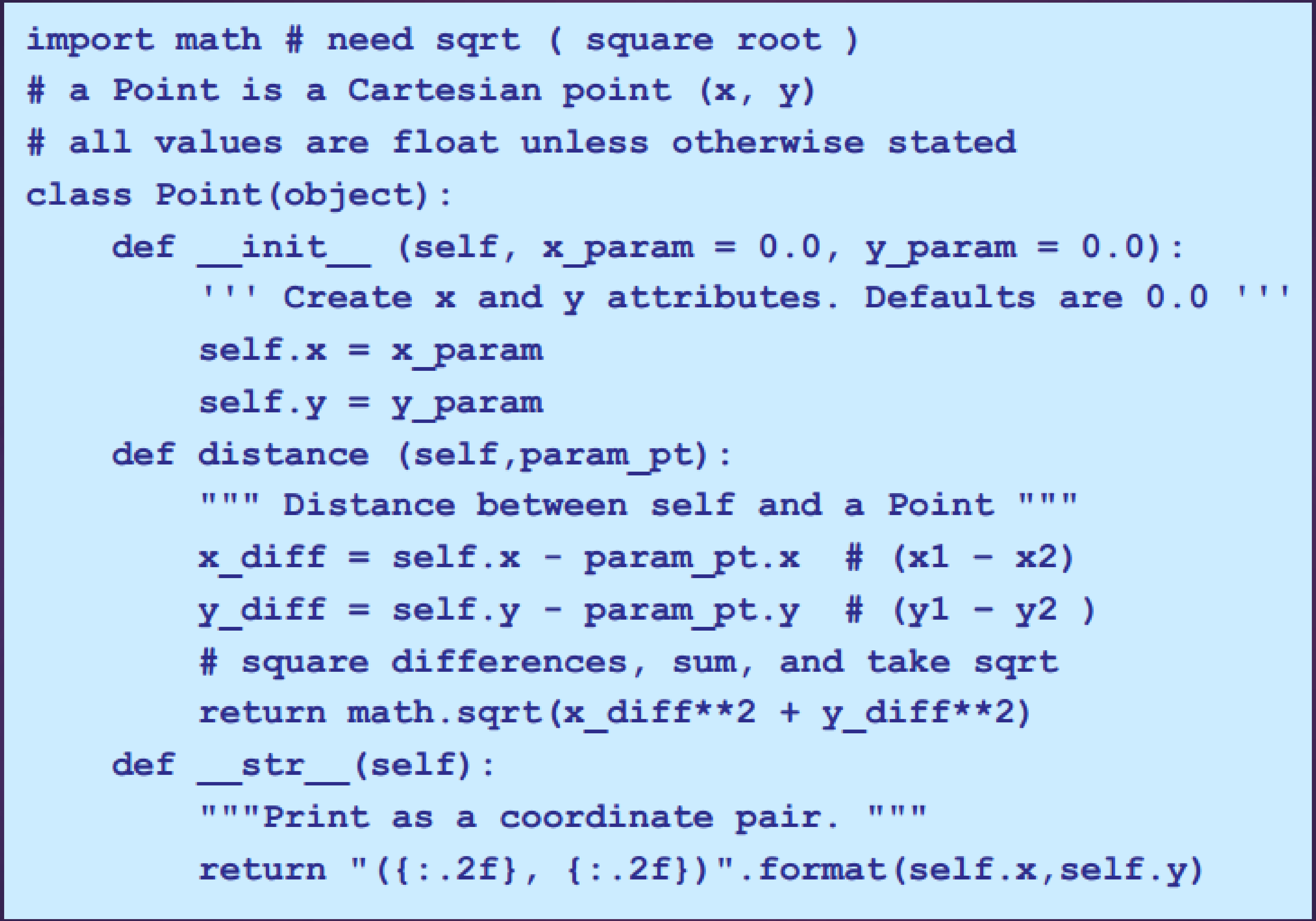
- If you don't provide a constructor, then only the default constructor is provided
- The default constructor does system stuff to create the instance, nothing more
- You cannot pass arguments to the default constructor
- By providing the constructor method, we ensure that every instance, at least at the point of construction, is created with the same contents
- This gives us some control over each instance.

__str__ and printing



```python
def __str__(self):
    # print "In __str__ method"
    return "{} {}, ID:{}".\
        format(self.first_name_str, self.last_name_str, self.id_int)
```

- When `print(my_inst)` is called, it is assumed, by Python,to be a call to “convert the instance to a string”, which is the job of the `__str__` method
- In the `__str__` method, my_inst is bound to self
- `__str__` ***must return a string***!
- The string returned by `__str__` is printed by the print function.



```python
import math # need sqrt ( square root )
# a Point is a Cartesian point (x, y)
# all values are float unless otherwise stated
class Point(object):
    def __init__ (self, x_param = 0.0, y_param = 0.0):
        ''' Create x and y attributes. Defaults are 0.0 '''
        self.x = x_param
        self.y = y_param
    def distance (self,param_pt):
        """ Distance between self and a Point """
        x_diff = self.x - param_pt.x  # (x1 - x2)
        y_diff = self.y - param_pt.y  # (y1 - y2 )
        # square differences, sum, and take sqrt
        return math.sqrt(x_diff**2 + y_diff**2)
    def __str__(self):
        """Print as a coordinate pair. """
        return "({:.2f}, {:.2f})".format(self.x,self.y)
```

Python operator

```
>>> 'he' + 'llo'
'hello'
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> 2+4
6
```

```
>>> 'he'.__add__('llo')
'hello'
>>> [1,2].__add__([3,4])
[1, 2, 3, 4]
>>> int(2).__add__(4)
6
```

Operator + is defined for multiple classes; it is an overloaded operator.

- For each class, the definition—and thus the meaning—of the operator is different.
  - integer addition for class `int`
  - list concatenation for class `list`
  - string concatenation for class `str`
- How is the behavior of operator + defined for a particular class?

Class method `__add__()` implements the behavior of operator + for the class

When Python evaluates

```
object1 + object 2
```

… it first translates it to method invocation …

```
object1.__add__(object2)
```

… and then evaluates the method invocation

In Python, all expressions involving operators are translated into method calls

```
>> '!'.__mul__(10)
'!!!!!!!!!!'
>>> [1,2,3].__eq__([2,3,4])
False
>>> int(2).__lt__(5)
True
>>> 'a'.__le__('a')
True
>>> [1,1,2,3,5,8].__len__()
6
```

```
>>> [1,2,3].__repr__()
'[1, 2, 3]'
>>> int(193).__repr__()
'193'
>>> set().__repr__()
'set()'
```

Operator	Method
x + y	x.__add__(y)
x – y	x.__sub__(y)
x * y	x.__mul__(y)
x / y	x.__truediv__(y)
x // y	x.__floordiv__(y)
x % y	x.__mod__(y)
x == y	x.__eq__(y)
x != y	x.__ne__(y)
x > y	x.__gt__(y)
x >= y	x.__ge__(y)
x < y	x.__lt__(y)
x <= y	x.__le__(y)
repr(x)	x.__repr__()
str(x)	x.__str__()
len(x)	x.__len__()
<type>(x)	<type>.__init__(x)

Overloading Operator +

To get this behavior

```
>>> a = Point(3,4)
>>> b = Point(1,2)
>>> a+b
Point(4, 6)
```

```
>>> a = Point(3,4)
>>> b = Point(1,2)
>>> a.__add__(b)
Point(4, 6)
```

method __add__() must be implemented and added to class Point

__add__() should return a new Point object whose coordinates are the sum of the coordinates of a and b

```
class Point:

    # other Point methods here

    def __add__(self, point):
        return Point(self.x+point.x, self.y+point.y)

    def __repr__(self):
        'canonical string representation Point(x, y)'
        return 'Point({}, {})'.format(self.x, self.y)
```

Overloading operator len

To get this behavior

```
>>> appts = Queue()
>>> len(appts)
0
```

```
>>> appts = Queue()
>>> appts.__len__()
0
```

method __len__() must be implemented and added to class Queue

__len__() should return the number of objects in the queue

- i.e., the size of list self.q

```
class Queue:
    def __init__(self):
        self.q = []

    def isEmpty(self):
        return (len(self.q) == 0)

    def enqueue (self, item):
        return self.q.append(item)

    def dequeue(self):
        return self.q.pop(0)

    def __len__(self):
        return len(self.q)
```

We use the fact that len()
is implemented for class list

Private variable

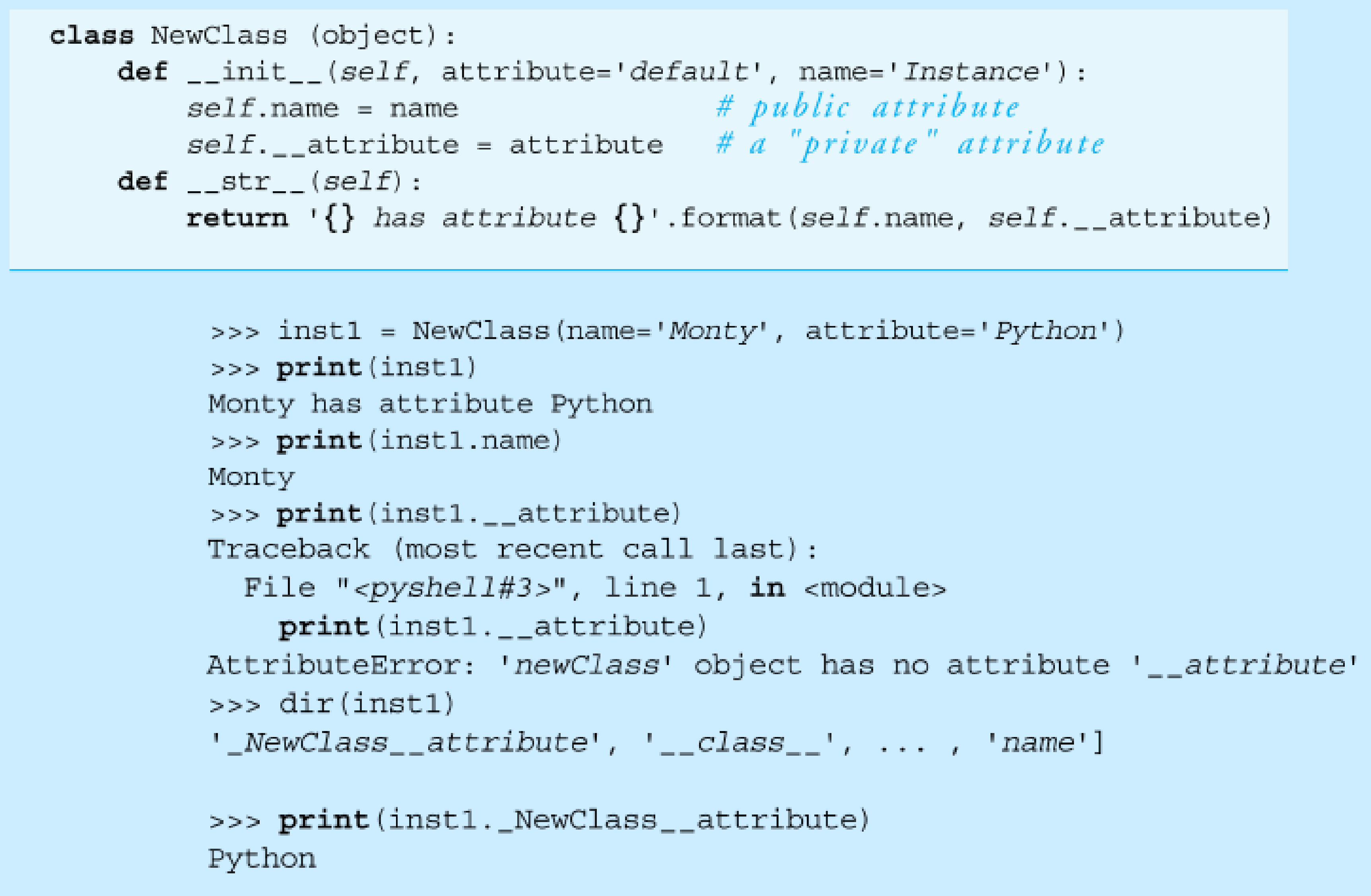
- many OOP approaches allow you to make a variable or function in an instance ***private***
- private means not accessible by the class user, only the class developer.
- there are advantages to controlling who can access the instance values

Privacy in python

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

- Python takes the approach "We are all adults here". No hard restrictions.
- Provides naming to avoid accidents. Use _ _ (double underscores) in front of any variable
- this ***mangles*** the name to include the class, namely __var becomes _class_ _var
- still fully accessible, and the _ _dict_ _ makes it obvious

Example



```
class NewClass (object):
    def __init__(self, attribute='default', name='Instance'):
        self.name = name                  # public attribute
        self.__attribute = attribute      # a "private" attribute
    def __str__(self):
        return '{} has attribute {}'.format(self.name, self.__attribute)
```

```
>>> inst1 = NewClass(name='Monty', attribute='Python')
>>> print(inst1)
Monty has attribute Python
>>> print(inst1.name)
Monty
>>> print(inst1.__attribute)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(inst1.__attribute)
AttributeError: 'newClass' object has no attribute '__attribute'
>>> dir(inst1)
'_NewClass__attribute', '__class__', ... , 'name']

>>> print(inst1._NewClass__attribute)
Python
```