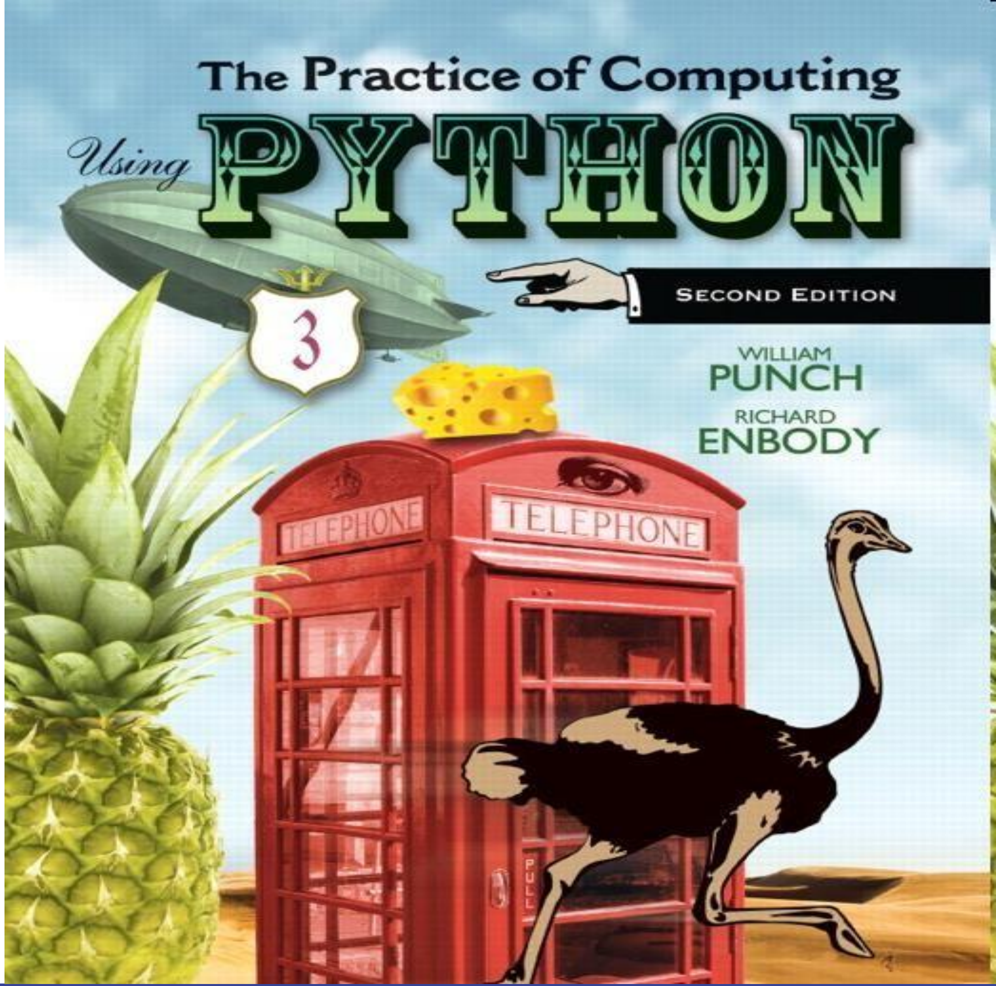
FAKULTAS ILMU KOMPUTER
UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

CSGE601020 | Dasar-Dasar Pemrograman 1

Introduction to Classes

Intro to Classes



Live coding: Mahasiswa

```
class Mahasiswa:

    def __init__(self, nama, NPM): # __init    adalah special method
untuk  instantiation (membuat objek/instance dari class)
         self.nama = nama # nama adalah instance attribute
         self.NPM = NPM # NPM adalah instance attribute

    def menyapa(self): # menyapa() adalah method dari instance class Mahasiswa
         print(self.nama + " : Hallo!")

    def __str__(self):
         return "Mahasiswa, Nama: {}, NPM: {}".format(self.nama, self.NPM)

mhs1 = Mahasiswa("Budi", "1234567890")
print(mhs1.nama)
print(mhs1.NPM)
print(mhs1)

mhs1.menyapa()
```

Object and Class

Objects and Programs

- You have learned how to structure your programs by decomposing tasks into functions.
    - Experience shows that it does not go far enough. It is difficult to understand and update a program that consists of a large collection of functions.
- To overcome this problem, computer scientists invented **object-oriented programming (OOP)**, a programming style in which tasks are solved by collaborating objects.
- Each object has its own set of data, together with a set of methods that act upon the data.

Objects and Programs

- You have already experienced this programming style when you used strings, lists, and file objects. Each of these objects has a set of methods.
- For example, you can use the `append()` method to operate on list objects.

Python Classes

- A class describes a set of objects with the same behavior.
    - For example, the `str` class describes the behavior of all strings.
    - This class specifies how a string stores its characters, which methods can be used with strings, and how the methods are implemented.
    - For example, when you have a `str` object, you can invoke the `upper` method:

```
"Hello, World".upper()
```

str object Method of class str

Python Classes

- In contrast, the `list` class describes the behavior of objects that can be used to store a collection of values.

- This class has a different set of methods.

- For example, the following call would be illegal—the `list` class has no `upper()` method.

```
["Hello", "World"].upper()
```

- However, `list` has a `pop()` method, and the following call is

```
["Hello", "World"].pop()
```

Objects

- An *object* can be considered as an active entity that knows stuff and can do stuff.
- More precisely, an object consists of:

1. A collection of related information (→ properties).
2. A set of operations to manipulate that information (→ behaviors).

Objects

- A class defines the properties and behaviors for objects.
- An object is an instance of a class.
- We can create many instances of a class.
- Creating an instance of a class is referred to as *instantiation*.
- A class is a description of what its instances will know and do.

Objects

- The information is stored inside the object in instance variables / data fields.
- The operations, called *methods*, are functions that “live” inside the object.
- Collectively, the instance variables and methods are called the *attributes* of an object.

Everything in Python is an **object**

- For example, [1, 2, 3] and "abc" are **objects**
- Each object has some number of **attributes** (e.g., *nama, NPM*)
- Object responds to **methods** (e.g., *menyapa()*)

Responding to "commands"

- As a set of interacting objects, each object responds to "commands"
- The interaction of objects via commands makes a **high level description** of what the program is doing

Class versus object (1)

- The analogy of the cookie cutter and a cookie.

Class versus object (2)

- You define a class as a way to generate object (instances of the class).
- The structure of an object starts out the same, as dictated by the class.
- The objects respond to the commands defined as part of the class.

Standard Class Names

The standard way to name a class in Python is called ***CapWords***:

- Each word of a class begins with a Capital letter
- no underlines
- sometimes called ***CamelCase***
- makes recognizing a class easier

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 stand-alone function.
- The first parameter of a method is *always* named **`self`**, which is a reference to the object on which the method is acting.

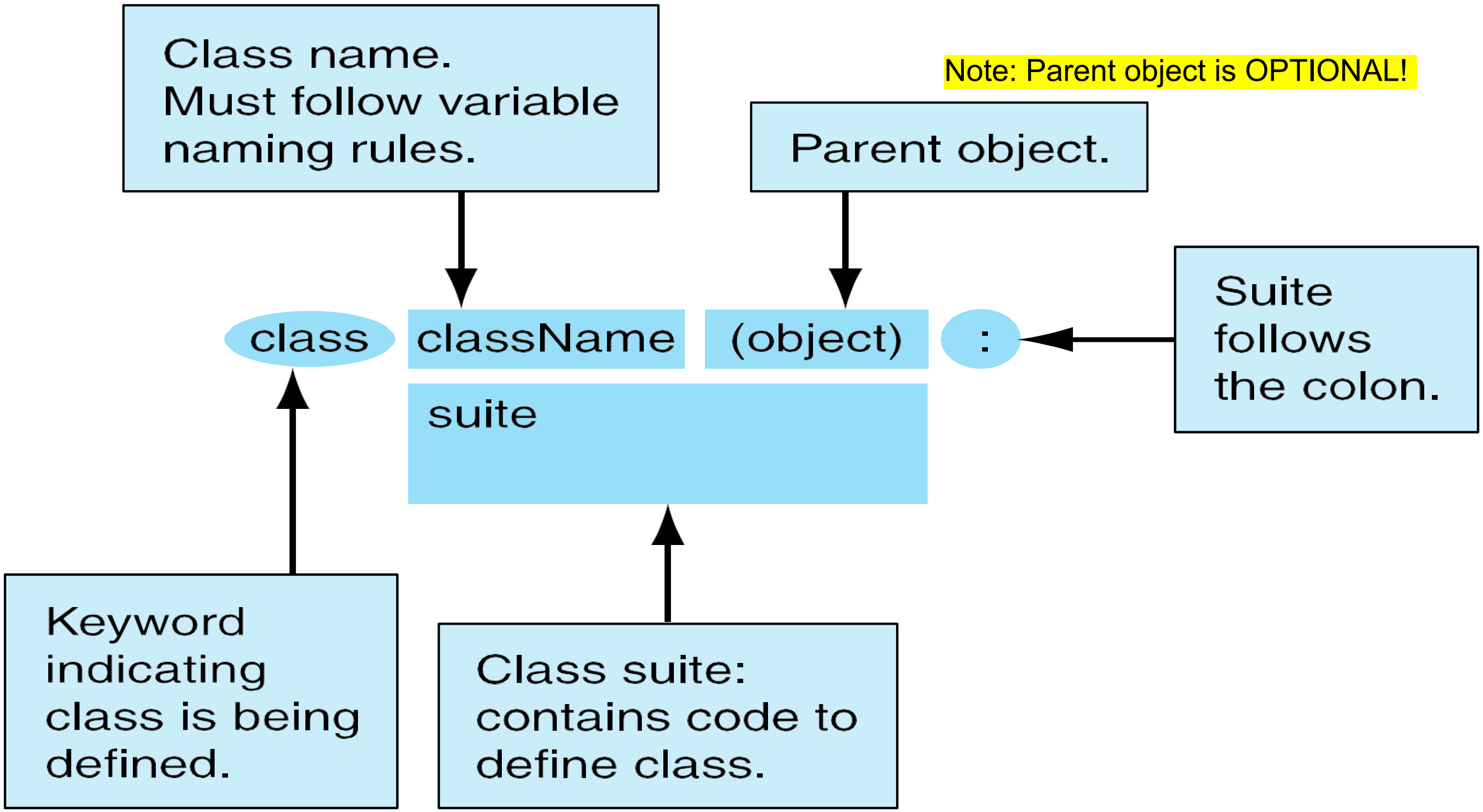


FIGURE 11.2 The basic format of a class definition.

Example

```
# circle.py
import math
class Circle:
    def __init__(self, radius = 1):
        self._radius = radius

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

    def getPerimeter(self): # harusnya get_perimeter
        return 2 * self._radius * math.pi

    def getArea(self): # harusnya get_area
        return math.pi * (self._radius ** 2)

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

```
>>> 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
```

```python
# testCircle.py
from circle import Circle

def main():
    # Create a circle with radius 1
    circle1 = Circle()
    print("The area of the circle of radius {} is
    {:.2f}."
        .format(circle1._radius, circle1.getArea()) )

    # Create a circle with radius 25
    circle2 = Circle(25)
    print("The area of the circle of radius {} is
    {:.2f}."
        .format(circle2._radius, circle2.getArea()) )

    # Modify circle radius
    circle2.setRadius(100)
    print("The area of the circle of radius {} is
    {:.2f}."
        .format(circle2._radius, circle2.getArea()) )

main()
```

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

UML Class Diagrams

- The illustration of class templates and objects can be standardized using UML (Unified Modeling Language) notation.
- *UML class diagrams* is language independent; that is, other programming languages, such as Java and C++, use this same modeling and notation.

UML Class Diagrams

- In UML class diagrams, data fields are denoted as:

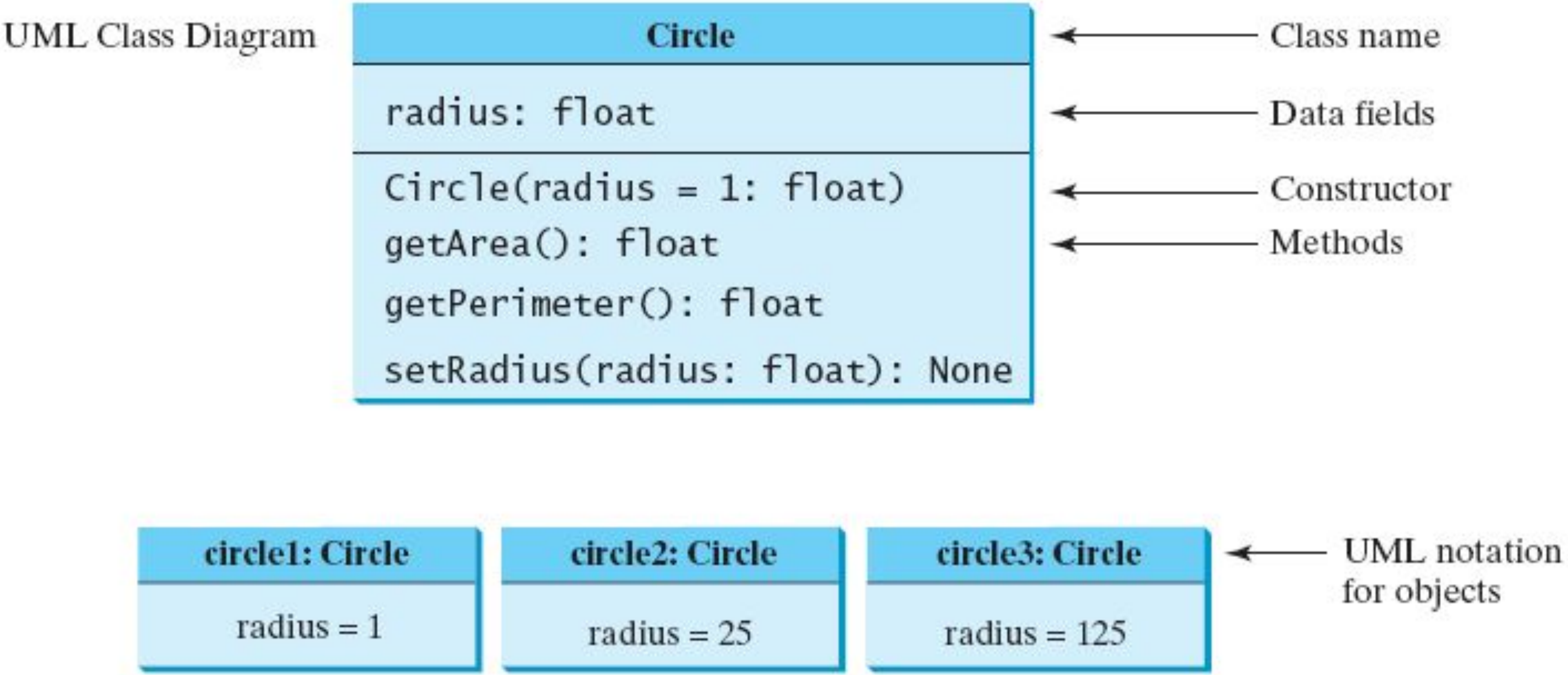
dataFieldName: dataFieldType

- Constructors are shown as:

ClassName(parameterName: parameterType)

- Methods are represented as:

methodName(parameterName: parameterType): returnType


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

OOP
(Object Oriented Programming)

OOP

- Object Oriented Programming (OOP) is a way to program in "objects"
- A program becomes:
    - less a list of instructions
    - more a set of objects and how they interact

OOP

- An object has a unique identity, state, and behavior.
  - An object's *identity* is like a person's Social Security number or NIK. Python automatically assigns each object a unique id for identifying the object at runtime.
  - An object's *state* (also known as its *properties* or *attributes*) is represented by variables, called *data fields*. A circle object, for example, has a data field **radius**, which is a property that characterizes a circle. A rectangle object has the data fields **width** and **height**, which are properties that characterize a rectangle.
  - Python uses methods to define an object's *behavior* (also known as its *actions*).

OOP

- Methods are defined as functions.
- You make an object perform an action by invoking a method on that object. For example, you can define methods named **getArea()** and **getPerimeter()** for circle objects. A circle object can then invoke the **getArea()** method to return its area and the **getPerimeter()** method to return its perimeter.

OOP principles

- ***modularity***: making multiple modules first and then linking and combining them
- ***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.

Constructor

Constructor

- When a class is defined, a function is made *with the same name as the class*
- This function is called the *constructor*. By calling it, you can **create an instance** of the class
- Constructor is called by using the name of the class as a function call (by adding () after the class name). Example:

```
m = Mahasiswa()
```

- Constructor provides a class designer the opportunity to set up the instance with variables, by assignment

defining the 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:

```
def __init__(self, nama,
      NPM):  self.nama = nama
      self.NPM = NPM
```

- 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.

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.

Every class should have: __init__

- 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.

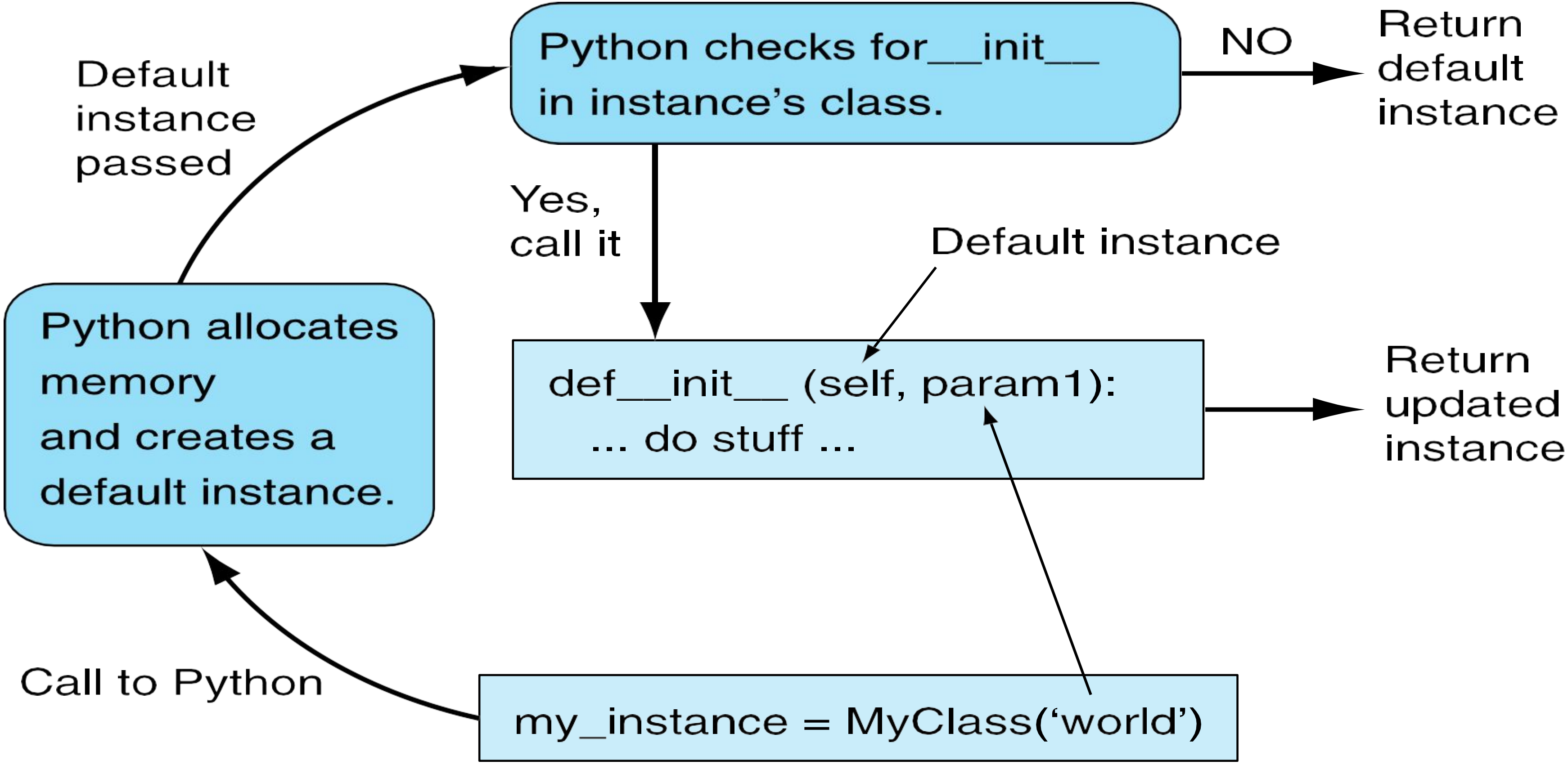


FIGURE 11.6 How an instance is made in Python.

Attributes (Variables)

Class components

- Each class has potentially two aspects:
  - the **data or attributes** (types, number, names) that each instance might contain; and
  - the **commands or methods** that each instance can respond to.

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 function
- it is part of the object, either directly or by that object being part of a class

Examples

`print(my_instance.my_var)`

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 parenthesis at the end of the reference

How to make an instance attribute

- Once an object is made, the data is made the same way as in any other Python situation, by assignment
- Any object can thus be augmented by adding a variable

```
my_instance.someattribute = 'hello'
```

dir() function

The `dir()` function lists all the attributes of a class or an instance

- you can think of these as keys in a dictionary

New attribute shown in dir

```
dir(my_instance)

['_class_', '_delattr_', '_dict_', '_doc_', '_format_',
'__getattribute ', ' hash ', ' init ', ' module ', ' new ',
'__reduce ', ' reduce_ex ', ' repr ', ' setattr ', ' sizeof ',
'__str ', ' subclasshook ', ' weakref ', someattribute]
```

Class attribute vs instance attribute

- **Class attributes:**
    - They belong to the class itself, so they will be shared by all the instances. All objects refer to single copy of the class attribute
    - They are defined in the class body parts
- **Instance attributes**:
    - They belong to object (instance of the class). Every object has its own copy of the instance attribute

```python
class Mobil:
    roda=4

    def __init__(self, merk=None, seri=None, warna = None):
        self.merk  = merk
        self.seri  = seri
        self.warna = warna

    def __str__(self):
        return "merk: " + self.merk + " seri: " + self.seri + "warna: " + self.warna

m1= Mobil("Toyota", "Avanza", "hitam")
m2 = Mobil("Honda", "Jazz", "kuning")

#cetak class attribute
print(m1.roda)
print(m2.roda)

#cetak instance attribute
print(m1.merk)
print(m2.merk)
```

Class Attribute

Instance Attribute

Live coding: Buat class MatkulFasilkom

```
.... :

    def_init_(self, nama_matkul):
        ....

    def cetak(self): # cetak, misalnya, "Matkul
    DDP"
        ....

ddp                         =
MatkulFasilkom('DDP')   ppw
=     MatkulFasilkom('PPW')
ddp.cetak()
ppw.cetak()
```

Live coding: Buat class MatkulFasilkom

```
class MatkulFasilkom :

    def_init_(self, nama_matkul):
        self.nama_matkul =
        nama_matkul

    def cetak(self): # cetak, misalnya, "Matkul
        DDP"  print("Matkul", self.nama_matkul)

ddp                        =
MatkulFasilkom('DDP')   ppw
=     MatkulFasilkom('PPW')
ddp.cetak()
ppw.cetak()
```

pass keyword

Remember, `pass` does nothing

- by making the suite of a class undefined using `pass`, we get only those things that Python defines for us automatically
- In other words, `pass` indicates empty suit

Live coding: Default attributes

```
class MyClass:
    pass

print(dir(MyClass))

my_instance = MyClass()
print(type(my_instance)
)

print(dir(my_instance))
```

Instance knows its class

- Because each instance has as its type the class that it was made from, an instance remembers its class
- This is often called the ***instance-of*** relationship
- stored in the __class__ attribute of the instance

Live coding

```
class MyClass:
    pass

my_instance = MyClass()
print(my_instance.__class__
)  print(type(my_instance))
```

```
>>> class MyClass (object):
        pass

>>> my_instance = MyClass()

>>> MyClass.class_attribute = 'hello'
>>> print(MyClass.class_attribute)
hello
>>> dir(MyClass)
['__class__', ..., 'class_attribute']

>>> my_instance.instance_attribute = 'world'
>>> print(my_instance.instance_attribute)
world
>>> dir(my_instance)
['__class__', ,..., 'class_attribute', 'instance_attribute']

>>> print(my_instance.class_attribute)
hello
>>> print(MyClass.instance_attribute)

Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print MyClass.instance_attribute
AttributeError: type object 'MyClass' has
no attribute 'instance_attribute'
```

Scope

It works differently in the class system, taking advantage of the ***instance-of*** relationship

Part of the Object Scope Rule

The first two rules in object scope are:

1. **First**, look in the object itself
2. If the object attribute is **not found**, look up to the class and search for the attribute there.

```
>>> class MyClass (object):
        pass

>>> inst1 = MyClass()
>>> inst2 = MyClass()
>>> inst3 = MyClass()
>>> MyClass.class_attribute = 27
>>> inst1.class_attribute = 72
>>> print(inst1.class_attribute)
72
>>> print(inst2.class_attribute)
27
>>> print(inst3.class_attribute)
27
>>> MyClass.class_attribute = 999
>>> print(inst1.class_attribute)
72
>>> print(inst2.class_attribute)
999
>>> print(inst3.class_attribute)
999
```

Methods

Live coding: MyClass

```
class MyClass:
    my_class_attr = "value of class attr"

    def my_method(self, param1):
        print("Param1:", param1)
        print("Object", str(self))
        self.my_instance_attr =
        param1

my_instance1 =
MyClass()  my_instance2
= MyClass()

my_instance1.my_method("string of
my_instance1")  my_instance2.my_method("string
of my_instance2")

print(my_instance1.my_instance_attr)
print(my_instance2.my_instance_attr)
```

method versus 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

- The difference is that methods are functions tied to a particular object

difference in calling

functions are called without the context of an object

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):
        ...
```

more on self

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

Live coding: self

```
class MyClass:
    my_class_attr = "value of class attr"

    def my_method(self, param1):
        print("Param1:", param1)
        print("Object", str(self))
        self.my_instance_attr =
        param1

my_instance1 = MyClass()
my_instance1.my_method("string of my_instance1") # my_instance1
is  passed as the first argument self of my_method
```

self is bound for us

- when a dot 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`.
- The method can be written generically, dealing with calling objects through `self`

Example

```
import math

class Point(object):
    def __init__(self, x_param = 0.0, y_param = 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)
        return math.sqrt(x_diff**2 + y_diff**2)

    def sum (self, param_pt):
    """ Vector Sum of self and a Point, return a Point instance """
        new_pt = Point()
        new_pt.x = self.x + param_pt.x
        new_pt.y = self.y + param_pt.y
        return new_pt
```

```
>>> p1 = Point(2.0,4.0)        # create a point with x and y values specified
>>> p2 = Point()               # create a point with default values
>>> print(p1.distance(p2))     # find and print the distance
4.47213595499958
>>> p3 = p1.sum(p2)            # calculate the sum and then print it
>>> print(p3.x, p3.y)
2.0 4.0
>>>
```

Example

```
import math

class Point(object):
    def      init     (self, x_param = 0.0, y_param =
        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)
        return math.sqrt(x_diff**2 + y_diff**2)

    def sum (self, param_pt):
    """ Vector Sum of self and a Point, return a Point instance """
        new_pt = Point()
        new_pt.x = self.x + param_pt.x
        new_pt.y = self.y + param_pt.y
        return new_pt

    def __str__(self):
    """Print as a coordinate pair . """
       print("called the str method")
       return "({:.2f}, {:.2f})".format(self.x, self.y)
```

```
>>> p1 = Point(2.0, 4.0)
>>> print(p1)
called the __str__ method
(2.00, 4.00)
```

Example

```
import math

class Point(object):
      def       init      (self, x_param = 0.0, y_param =
            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)
            return math.sqrt(x_diff**2 + y_diff**2)

      def sum (self, param_pt):
      """ Vector Sum of self and a Point, return a Point instance """
            return Point(self.x + param_pt.x, self.y + param_pt.y)

      def __str__(self):
      """Print as a coordinate pair . """
           print("called the str method")
           return "({:.2f}, {:.2f})".format(self.x, self.y)
```

Python Standard Methods

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__
- by using these methods, we "fit in" to the normal Python flow

str , printing

```
def __str__(self):
    return "Mahasiswa, Nama: {}, NPM: {}".format(self.nama, self.NPM)
```

- When `print(my_inst)` called, it is assumed, by Python, to be a call to "convert the instance to a string", which is the str method
- In the method, `my_inst` is bound to `self`, and printing then occurs using that instance.
- It must return a string!

Destructor

You can construct, and you can destruct, using the method __del__.

Live coding: Class Mobil

```
class Mobil:
    roda=4

    def_init_(self, merk=None, seri=None, warna = None):
        self.merk = merk
        self.seri = seri
        self.warna = warna

    def_str_(self):
        return "merk: " + self.merk + " seri: " + self.seri + "warna: " + self.warna

    def __del__(self):
        print("objek mobil sudah dihapus")

m = Mobil("Toyota", "Avanza", "hitam")
print(m)
del m
print(m)
```

Now there are three groups

There are now three groups in our coding scheme:

- user
- programmer as class user
- programmer as class designer

Class designer

- The class designer is creating code to be used by other programmers
- In so doing, the class designer is making a kind of library that other programmers can take advantage of


PRIVATE
PROPERTY
NO
TRESPASSING

class namespaces are dicts

- the namespaces in every object and module are indeed a dictionary
- that dictionary is bound to the special variable __dict__
- it lists all the local attributes (variables, functions) in the object

private variables in an instance

- 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

private variables in an instance

Attribute classification:

- Private attributes **should** only be used by the owner, i.e. inside of the class definition itself. They have two leading underscores __ as prefix.
- Public attributes can and should be freely used (inside or outside class definition).

Live coding: Private vs. public attributes

```
class MyClass:


  def __init__(self):
    self.my_public_attr = "Public"
    self.__my_private_attr = "Private"


  def print_private(self):
    print(self.__my_private_attr)


obj = MyClass()
print(obj.my_public_attr)
obj.print_private()
print(obj.__my_private_attr)
```

Live coding: Private vs. public attributes

```
class MyClass:

    def __init__(self):
        self.my_public_attr = "Public"
        self.__my_private_attr = "Private"

    def print_private(self):
        print(self.__my_private_attr)

obj = MyClass()
print(obj.my_public_attr)
obj.print_private()
print(obj._MyClass__my_private_attr)
```

privacy in Python

- Python takes the approach "We are all adults here". No hard restrictions.
- Provides naming to avoid accidents. Use __ double underlines in front of any variable
- This ***makes*** the name to include the class, namely __var becomes _class__var
- still fully accessible, and the __dict__ makes it obvious

Live coding

```
class MyClass:

    def __init__(self):
        self.my_public_attr = "Public"
        self.__my_private_attr = "Private"

    def print_private(self):
        print(self.__my_private_attr)

obj = MyClass()
print(obj. __dict__)
```

References

- Lim Yohanes Stefanus. Slide Mata Kuliah Dasar-Dasar Pemrograman 1 topik “Intro to Classes”. 2019.
- Ljubomir Perkovic. Introduction to Computing Using Python: An Application Development Focus. 2nd Edition. Wiley, 2015.
- William Punch & Richard Enbody. The Practice of Computing using Python, 3rd Edition. Addison-Wesley, 2017.
- Y. Daniel Liang. Introduction to Programming Using Python. Prentice Hall, 2013.
- John Zelle. Python Programming: An Introduction to Computer Science, 3rd Edition. Franklin, Beedle & Associates Inc., 2017.
- Mark Lutz. Learning Python, 5th Edition. O’Reilly Media. Copyright 2013 Mark Lutz, 978-1-4493-5573-9.

Thanks