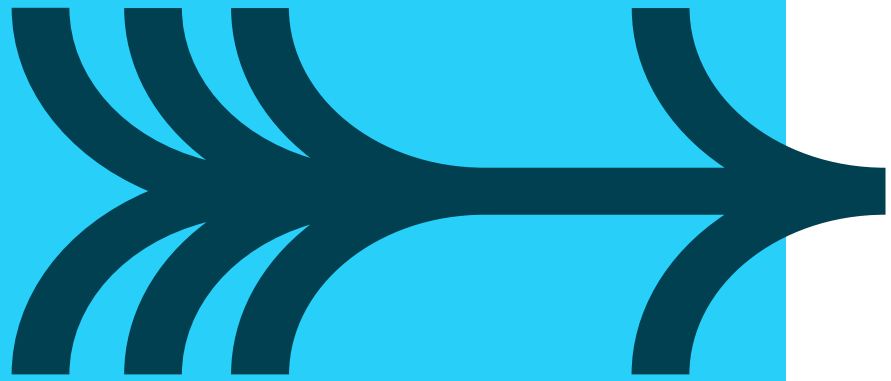
POWERING POTENTIAL

Object Orientation

Module 16

CLASSES AND OOP



Contents

Using objects

Calling a class creates a new *instance object*

- Invokes the constructor

```
from account import Account

some_account = Account(1000.00)
some_account.deposit(550.23)
some_account.deposit(100)
some_account.withdraw(50)
print(some_account.getbalance())

another = Account(0)

print(Account.numCreated)
print("object another is class",
      another.__class__.__name__)
```

```
1600.23
2
object another is class Account
```

A class is not a type!

Don't ask what type an object is, only ask what the object can do

```
hasattr(object, name)
```

This is known as *duck-typing*

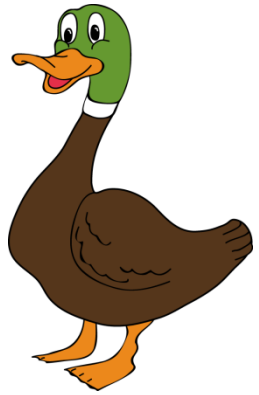
"If it walks like a duck, swims like a duck, and quacks like a duck ..."

```
if hasattr(x, '__str__'):
    val = str(x)
```

- In the example, we don't care what class **x** belongs to, only that it supports representation as a string

Based on the original concepts of object orientation

- Send/receive messages to/from an object
- *Signature-based* polymorphism



A little Python OO

- **A class is declared using `class`**
  - Membership is by *indentation*
- ***"class names use the CapWords convention"* – PEP008**
- **Methods are declared as functions within that class**
  - First argument passed is the object
  - The constructor is called `__init__`
  - The destructor is called `__del__`

→ Rarely required and unreliable

Classes are usually declared in a module

- File usually has same name as the class, with .py appended
- Simple example over…

Defining classes

- **The class statement**
  - Defines a class object

→ Public attributes are referenced by *Class.attribute*

  - Usually in a module with the same name as the class

```
class Account:
    numCreated = 0
    def __init__(self, initial):
        self._balance = initial
        Account.numCreated += 1

    def deposit(self, amt):
        self._balance += amt
        return

    def withdraw(self,amt):
        self._balance -= amt
        return

    def getbalance(self):
        return self._balance
```

account.py

Public class variable

Private class variable

Methods

Defining methods

- **Methods are functions defined within a class**
- **Conventions with underscores - reminder**
  - Names beginning with one underscore are private to a *module/class*
  - Names beginning with two underscores are private and mangled
  - Names surrounded by two underscores have a special meaning
  - Note: these do not guarantee privacy!
- **Object methods**
  - First argument passed to a method is the object

→ Usually called 'self', but can be anything

- **Class methods and attributes**
  - Defined within the class
  - Can be called on a class or object

Constructing an object

Python has two methods:

- __new__

→ Called when an object is created

→ First parameter is the class name

→ Return the constructed object

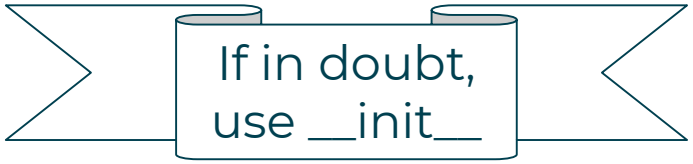
- __init__

→ Called when an object is initialised

→ First parameter is the object

→ An implicit return of the current object

- __new__ is called in preference



Which to use?

- Use __new__ only if constructing an object of a different class
- In most cases, use `__init__`

Special methods

- **A mechanism for operator and special function overloading**
- Assists with duck-typing
- **Function names start and end with two underscores**

py3

`__bool__(self)`	Return True or False
`__del__(self)`	Called when an object is destroyed
`__format__(self, spec)`	`str.format` support
`__hash__(self)`	Return a suitable key for dictionary or set
`__init__(self, args)`	Initialise an object
`__len__(self)`	Implement the `len()` function
`__new__(class, args)`	Create an object
`__repr__(self)`	Return a python readable representation
`__str__(self)`	Return a human readable representation

Operator overload special methods

- **All operators may be overloaded**
  - See the online documentation for a complete list

Return types vary

- Can return a `NotImplemented` object
- Examples:

`__add__`	+
`__sub__`	-
`__eq__`	==
`__ge__`	>=
`__lt__`	<
`__invert__`	~ (logical NOT)
`__getitem__(self,`*key*`)`	container element evaluation
`__setitem__(self,` *key*`,` *value*`)`	container element assignment

Special methods - example

```
class Date:

    def __init__(self, day=0, month=0, year=0):
        self._day   = day
        self._month = month
        self._year  = year

    def __str__(self):
        return "%02d/%02d/%d" % (
                self._day, self._month, self._year)

    def __add__(self, value):
        retn = Date(self._day, self._month, self._year)
        retn._day = retn._day + value
        retn._validate_date()
        return retn

today = Date(9, 10, 2015)
print(today)
tomorrow = today + 1
print(tomorrow)
```

Note private variable and method names starting with two underscores

```
09/10/2015
10/10/2015
```

Properties

Built-in property() creates an attribute

- property() has getter, setter, deleter, and docstring
- The appropriate method is called depending on the way the attribute is used

```
class Date:
    ...
    def mget(self):
        return self._day

    def mset(self, day):
        self._day = day

    mday = property(mget, mset)
```

Call the (default) getter method

```
day = today.mday
```

Call the setter method

```
today.mday = 6
```

Omitting the setter method means that the attribute is *read-only*

Properties and decorators

- **A decorator is a function name prefixed @**
  - The function will normally return another function
  - The decorator is followed by the function to be returned
- **Decorators are syntactic sugar, but commonly used**
  - Built-in property() is usually called using a decorator

```
class Date:
    ...
    @property
    def mday(self):
        return self._day

    @mday.setter
    def mday(self, day):
        self._day = day
```

Call the (default) getter method

```
day = today.mday
```

Call the setter method

```
today.mday = 6
```

Decorators - what's the point?

- **Decorators are part of Python function syntax**
  - Not specifically OO, but often used in OO contexts
  - Part of *metaprogramming*
- **One aim is to make code easier to read**
  - The decorator 'decorates' functionality

```
    def mget(self):
        return self._day
    ...
    mday = property(mget, mset)
```

Trailing property() call might be missed, or forgotten. When does it get executed?

```
    @property
    def mday(self):
        return self._day
    ...
```

Method is bound to the attribute name, and bound to @property.

Class methods

There are several ways to achieve this

- Using a dummy class wrapper
- Using the classmethod built-in as a decorator (preferred)

→ The class method itself

```
    _count = 0
...
    @classmethod
    def get_count(cls):
        return Date._count
```

The class name is passed implicitly

→ The user of the class

```
from date import Date
...
counter = Date.get_count()
```

Inheritance

Use attributes and methods from a parent class

- Important OO concept

→ Python supports multiple inheritance - not often needed

- Attributes and methods not supplied in the derived class will be inherited from the base class
- Common to derive our own classes from Python's own

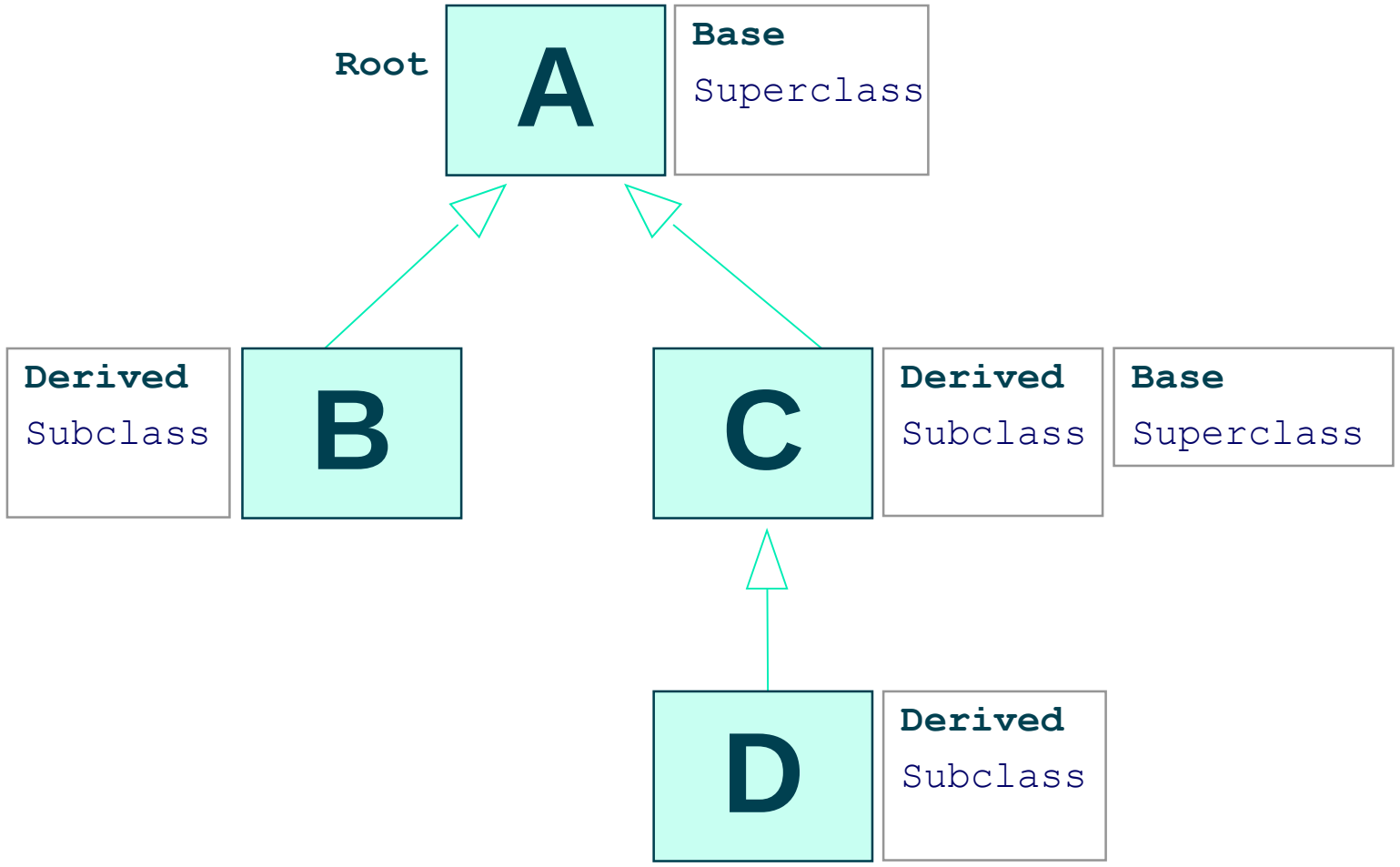
→ Multithreading

→ Exceptions

→ etc.

```
class DerivedClassName(base_classes):

    def __init__(self, arguments):
        base_class.__init__(self, arguments)

Other methods…
```

Inheritance terminology



Inheritance scope

- **Attributes are either “public” or “private”**
  - No equivalent of “protected”
  - This includes methods as well as data items
  - Enforced by the leading two-underscores rule
  - Base and derived classes can share the same module

  → Can share attributes privately that are prefixed with a single underscore
- **Public attributes of the base class can be called on an object of the derived class**
  - Also applies to __special__ methods

Inheritance example

```
class Person:
    def __init__(self, name, gender):
        self._name = name
        self._gender = gender.upper()

    def __str__(self):
        return "Name: " + self._name+ \
               " Gender: " + self.__gender
```

```
from person import Person

class Employee(Person):

    def __init__(self, name, gender, dept):
        super().__init__(name, gender)
        self._dept = dept
...
```

User's view

```
from employee import Employee

me = Employee("Fred Bloggs",
              'm', 'IT')
print(me)
```

This calls the parent class special method

`super()` syntax

Some helper built-in functions

isinstance(*object*, *classinfo*)

- Returns True if *object* is of class *classinfo*

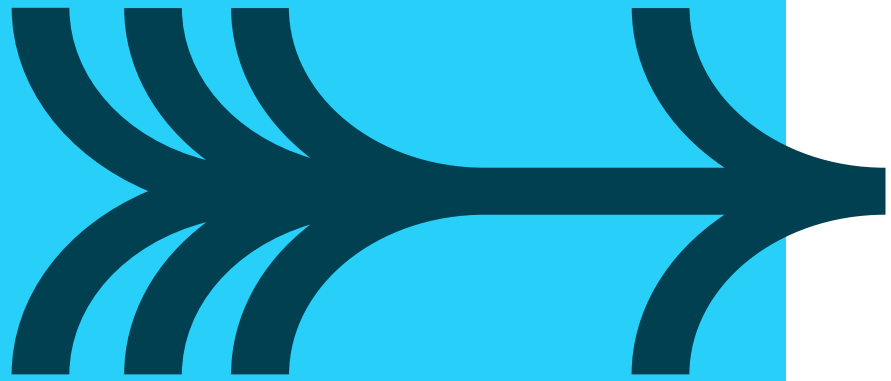
issubclass(*class*, *classinfo*)

- Returns True if *class* is a derived class of *classinfo*

```
from employee import Employee
from person import Person

me = Employee("Fred Bloggs", 'm', 'IT')

if isinstance(me, Employee):
    print(me, "isa Employee!")

if isinstance(me, Person):
    print(me, "isa Person!")

if issubclass(Employee, Person):
    print("Employee is a subclass of Person")
```

All these conditions return True (based on the inheritance example)

SUMMARY



- **Classes vs. objects**
- A class is a user defined data type
- An object is an instance of a class
- Objects have identity
- To achieve behavior we call an operation on an object
- The operations on an object are defined by its class
- **Encapsulation**
- Separates interface from implementation
- Publicly accessible operations
- Privately maintained state

Metaclasses and ABC

- **A metaclass is a class for creating other classes**
  - The syntax for metaclasses changed at Python 3
- **Abstract Base Classes**
  - Classes that cannot be directly instantiated
  - Created metaclass ABCMeta and decorator abstractmethod

```
from abc import *

class Vehicle(metaclass = ABCMeta):
    @abstractmethod
    def getReg(self):
        pass

class Car(Vehicle):
    def getReg(self):
        print("Car isa Vehicle")
```

```
beepbeep = Car()
beepbeep.getReg()
```

```
Car isa Vehicle
```

```
NoGo = Vehicle()
```

```
Can't instantiate abstract class Vehicle...
```