Advanced Python3: Object-oriented programming, databases and visualisation

Alexandra Diem
Simula Research Laboratory

simula consulting

simula

L01: Object-oriented programming

simula consulting

simula

Python is an interpreted, high-level, multi-paradigm language

Interpreted means that Python code is executed at runtime.

High-level means that Python is designed to be easily human readable, meaning that anyone with any programming experience should understand Python code.

Multi-paradigm means that it supports more than one programming paradigm:
- functional programming
- procedural programming
- object-oriented programming

So, what's an object?

Everything is an object!

Everything is an object

simula consulting

simula

Example: Skiers and mountains

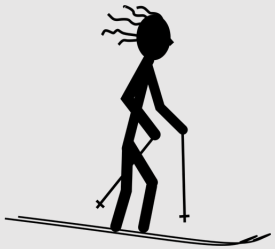
simula consulting

simula

Example: Skiers and mountains

Skier *can*:
- move
- put skins on skis
- remove skins on skis
- trigger avalanche

Skier *has*:
- name
- equipment
- ability level



Mountain *can*:
- avalanche

Mountain *has*:
- snow condition
- slope steepness
- weather
- avalanche danger

simula consulting

simula

Example: Skiers and mountains

Skier *can*:
- move
- put skins on skis
- remove skins on skis
- trigger avalanche

Skier *has*:
- name
- equipment
- ability level

Mountain *can*:
- avalanche

Mountain *has*:
- snow condition
- slope steepness
- weather
- avalanche danger

objects

simula consulting

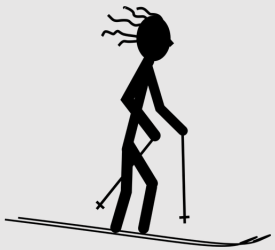
simula

Example: Skiers and mountains

Skier *can*:
- move
- put skins on skis
- remove skins on skis
- trigger avalanche

Skier *has*:
- name
- equipment
- ability level

classes



Mountain *can*:
- avalanche

Mountain *has*:
- snow condition
- slope steepness
- weather
- avalanche danger

simula consulting

simula

Example: Skiers and mountains

Objects, classes, instances

Disclaimer: These are not formal definitions because, sadly, computer science is not rigorously defined.

An **object** is an abstract concept that describes any*thing*.

A **class** is a description of the features that identify an object.

An **instance** is a specific implementation of a class. *Note: Some (many) will define an instance as an object described by a class, but I prefer to avoid this circular dependence and only refer to object as the abstract concept.*

Example: Person

A person *can*:
- sleep
- wake
- work
- greet another
- have a birthday

A person *has*:
- name
- age
- wake state
- working

simula consulting

simula

Objects, classes, instances

```
# person.py

class Person(object):
      pass
```

```
from person import Person

alice = Person()
bob = Person()

alice.name = ‘Alice’
alice.age = 20
alice.state = ‘awake’
alice.working = False
bob.name = ‘Bob’
bob.age = 34
bob.state = ‘awake’
bob.working = False
```

Object **Person**

simula consulting

simula

Objects, classes, instances

```
# person.py

class Person(object):

  def __init__(self, name, age,
               state='awake',
               working=False):
    self.name = name
    self.age = age
    self.state = state
    self.working = working
```

```
from person import Person

alice = Person('Alice', 20)
bob = Person('Bob', 34)
```

Object **Person**

simula consulting

simula

Example: Person

A person *can*:

- sleep
- wake
- work
- greet another
- have a birthday

affects

A person *has*:

- name
- age
- wake state
- working

simula consulting

simula

Objects, classes, instances

```
# person.py

class Person(object):

  def __init__(self, ...):
    self.name = name
    self.age = age
    self.state = state
    self.working = working

  def sleep(self):
    self.state = “asleep”
    self.working = False # side
effect from context

  def birthday(self):
    self.age += 1
```

```
from person import Person

alice = Person(‘Alice’, 20)
bob = Person(‘Bob’, 34)

alice.sleep()
bob.birthday()
alice.birthday()
alice.sleep()
```

Object **Person**

simula consulting

simula

Objects, classes, instances

```
# person.py

class Person(object):

  def __init__(self, ...):
    self.name = name
    self.age = age
    self.state = state
    self.working = working

  def greet(self, someone):
    print(f"{self.name} greets
          {someone.name}.")
```

```
from person import Person

alice = Person('Alice', 20)
bob = Person('Bob', 34)

alice.greet(bob)
[0]: Alice greets Bob.

bob.greet(alice)
[0]: Bob greets Alice.

print(alice)
[0]: <__main__.Person at
0x7fe144b35160>
```

Object **Person**

simula consulting

simula

Objects, classes, instances

```
# person.py

class Person(object):

  def __init__(self, ...):
    self.name = name
    self.age = age
    self.state = state
    self.working = working

  def __str__(self):
    return f"{self.name} is
      {self.age} years old.
      {self.name} is currently
      {self.state} and {'not  '
      if not self.working else
      ''}working."
```

```
from person import Person

alice = Person('Alice', 20)
bob = Person('Bob', 34)

print(alice)
[0]: Alice is 20 years old.
Alice is currently awake and
not working.

print(bob)
[0]: Bob is 34 years old. Bob
is currently awake and not
working.
```

Object **Person**

simula consulting

simula

Functions and methods

Disclaimer: These are not formal definitions because, sadly, computer science is not rigorously defined.

A **function** is a set of commands executed on arguments (objects). In Python it is specified using the def keyword.

A **method** looks like a function, but is bound to an object and belongs to the class specification. It always takes the keyword self as its first argument. *Note: Some (many) will also use the word 'function' when referring to methods, but it is never the other way around!*

Functions and methods

```
# person.py

class Person(object):

  def __init__(self, ...):
    self.name = name
    self.age = age
    self.state = state
    self.working = working

  def greet(self, someone):
    print(f"{self.name} greets
          {someone.name}.")
```

```
from person import Person

def greet(person0, person1):
  print(f"{person0.name}
  greets {person1.name}.")

alice = Person('Alice', 20)
bob = Person('Bob', 34)

greet(alice, bob) # Function
[0]: Alice greets Bob.

alice.greet(bob) # Method
[0]: Alice greets Bob.
```

Object **Person**

simula consulting

simula

Class and instance attributes

Disclaimer: These are not formal definitions because, sadly, computer science is not rigorously defined.

The value of a **class attribute** is shared by all instances of the same class. Mutating a class attribute on one instance results in the same mutation of the class attribute on all existing and new instances of the class. When a class attribute is redefined on an instance it turns into an instance attribute.

The value of an **instance attribute** is unique to each instance of a class.

Class and instance attributes

```
# person.py

class Person(object):

  clothing = [‘trousers’, ‘shirt’]

  def __init__(self, ...):
    self.name = name
    self.age = age
    self.state = state
    self.working = working
```

```
from person import Person

alice = Person(‘Alice’, 20)
bob = Person(‘Bob’, 34)

print(alice.clothing)
[0]: [‘trousers’, ‘shirt’]
print(bob.clothing)
[0]: [‘trousers’, ‘shirt’]

alice.clothing[0] = ‘skirt’
print(alice.clothing)
[0]: [‘skirt’, ‘shirt’]
print(bob.clothing)
[0]: [‘skirt’, ‘shirt’]
```

Object **Person**

simula consulting

simula

Private attributes

Private attributes that cannot be touched from outside the class definition are a fundamental feature of most object-oriented programming languages. They do not exist in Python.

However, *by convention only*, class or instance attributes whose names start with underscore(s) are to be considered private. This is usually used to indicate the expectation that these values never change.

Also by convention, such “private” variable are expected to be accessed by a getter-method.

Private attributes

```
# person.py

class Person(object):

  def __init__(self, ...):
    self._name = name
    self.age = age
    self.state = state
    self.working = working

  def get_name(self):
    return self._name
```

```
from person import Person

alice = Person('Alice', 20)
bob = Person('Bob', 34)

print(alice.get_name())
[0]: 'Alice'
print(bob._name)
[0]: 'Bob'

alice._name = 'Carol'
print(alice.get_name())
[0]: 'Carol'
```

Object **Person**

simula consulting

simula

Exercise: Deck of cards and players

Implement the classes **Card**, **Deck**, and **Player** as an interface to play a game of cards using the standard 52 cards deck (suits are spades, hearts, diamonds, clubs). Classes need appropriate attributes and methods to for typical actions of a card game (a card for example has a suit and a value, a player for example has a hand and can pick a card from the deck into their hand or discard a card from their hand).

Advanced: Implement a simple version of the card game **Hearts**, played by four players. Each round a random card is played by each player (they must follow suit if possible) from their hand. The trick is won by the highest card of the leading suit. The player with the fewest tricks wins the game.

Exercise: Deck of cards and players

Hint 1: It can be useful to choose an appropriate encoding for suits and special values jack, queen, king, ace into numbers using a dictionary

```
suits = {0: "spades", 1: "hearts", 2: "diamonds", 3: "clubs"}
vals = {1: "ace", 2: "2", 3: "3", 4: "4", 5: "5", 6: "6", 7: "7",
        8: "8", 9: "9", 10: "10", 11: "J", 12: "Q", 13: "K"}
```

Exercise: Deck of cards and players

Hint 2: You can shuffle a list using the function shuffle from the module random.

```
import random
lst = list(range(5))
print(lst)
[0]: [0, 1, 2, 3, 4]
random.shuffle(lst)
print(lst)
[0]: [3, 0, 2, 1, 4]
```

Inheritance and polymorphism

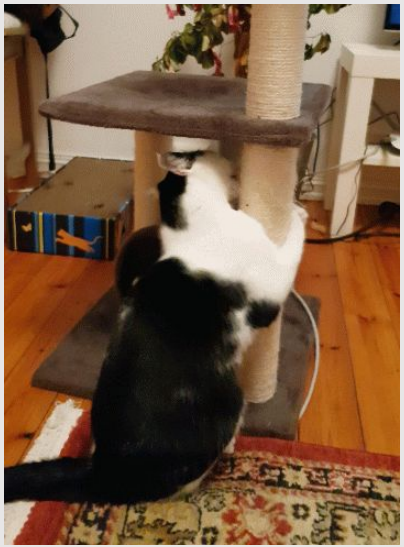
One of the major advantages of object-oriented programming is its inherent ability to avoid code duplication.

Inheritance is the derivation of a child class from one or many parent classes with whom it shares many common features.

Polymorphism is a Greek-derived compound word that stands for “many forms”. It means that a child class can reimplement (override) methods that are provided by the parent class.

Example: Pets

Cat

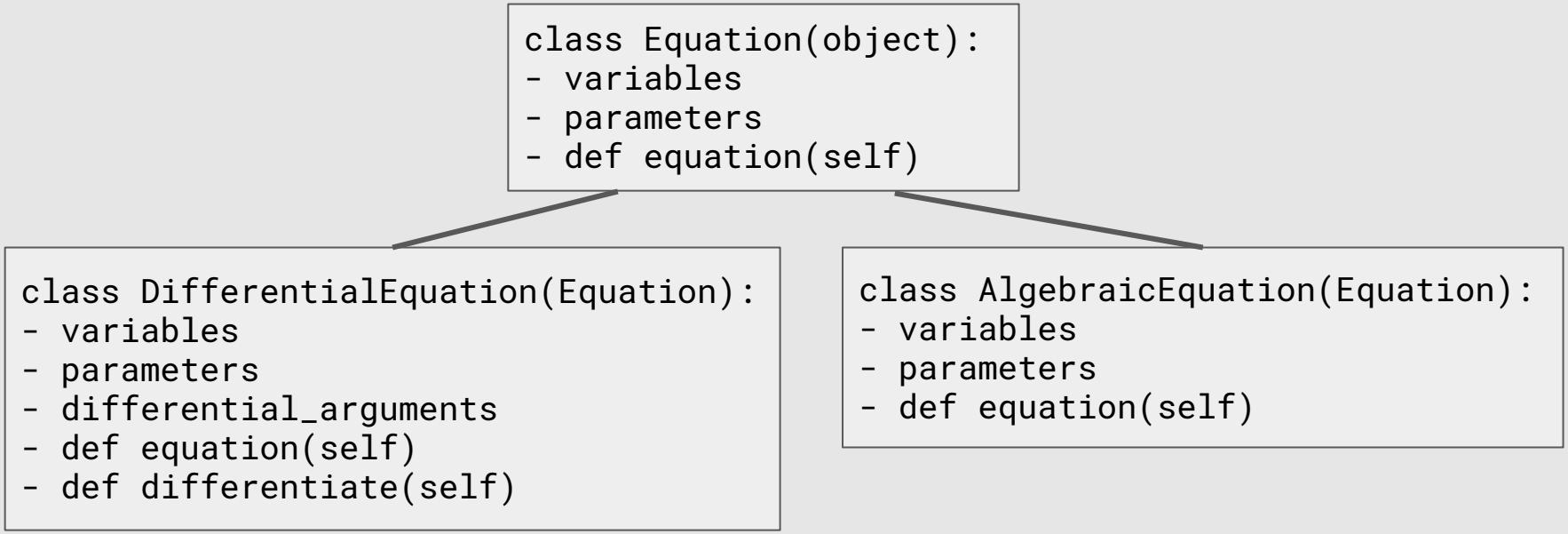


Dog

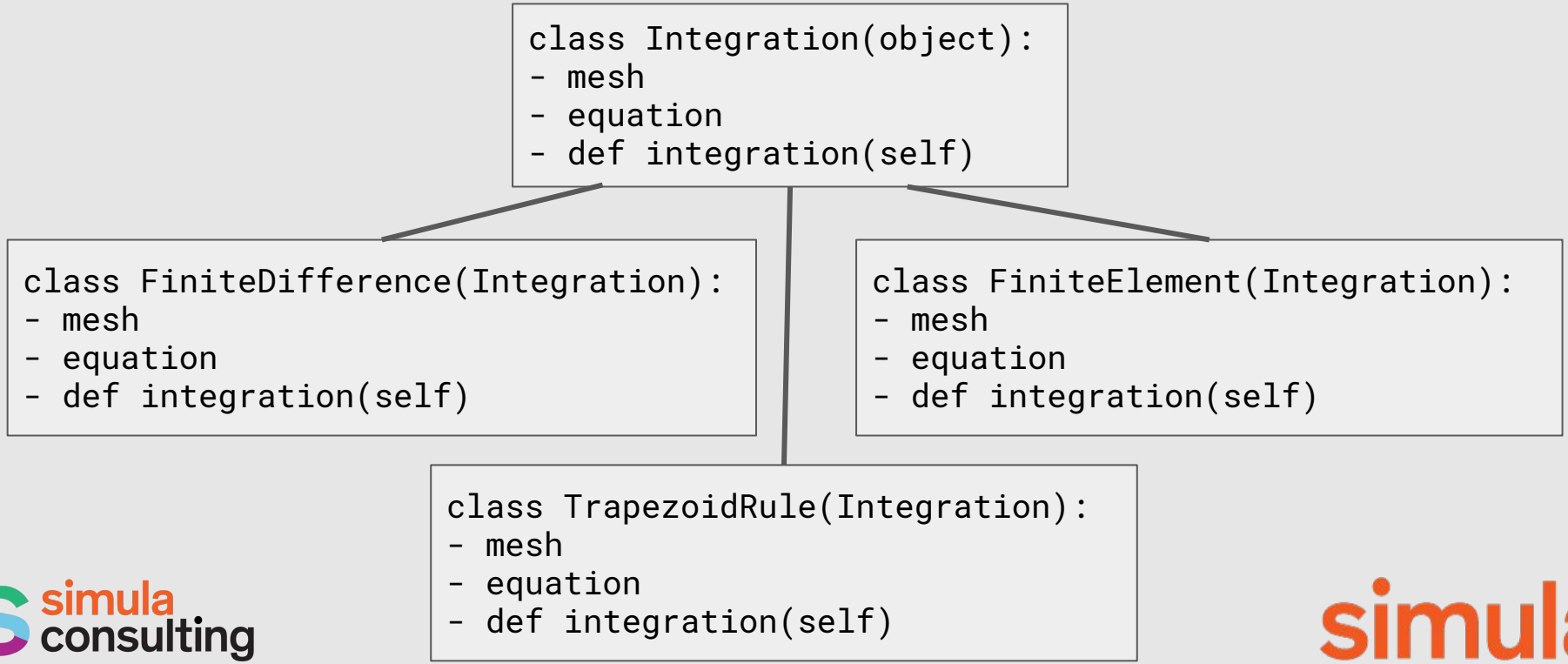
simula consulting

simula

Examples: Numerical problems/solvers



Examples: Numerical problems/solvers



simula consulting

simula

Exercise: Card games

Implement the class **CardGame** and two child classes **Hearts** and **OhHell**.

Hearts is played by 4 players, who are dealt 13 cards each. Each round a random card is played by each player (they must follow suit if possible) from their hand. The trick is won by the highest card of the leading suit. The player with the fewest tricks wins the game.

OhHell is played by 3+ players who are dealt as many cards as possible. Each player makes a prediction for the number of tricks they will get, otherwise a round is played the same way as in Hearts. The player who gets closest to their prediction wins (in case of a draw the player with more tricks wins).

Modules and packaging

A **module** is any file containing definitions and statements. Its functionality can be used by importing it

```
# person.py

class Person(object):

  def __init__(self, ...):
    self._name = name
    self.age = age
    self.state = state
    self.working = working

  def get_name(self):
    return self._name
```

```
from person import Person

alice = Person('Alice', 20)
bob = Person('Bob', 34)

print(alice.get_name())
[0]: 'Alice'
print(bob._name)
[0]: 'Bob'
```

simula consulting

simula

Modules and packaging

A **package** is a collection of modules that form a logical unit and whose *namespace* is organised by "dotted module names".

```
company/
  __init__.py
  person/
    __init__.py
    person.py
    employee.py
    customer.py
  invoice/
    __init__.py
    invoice.py
```

```
from company.person import Employee,
Customer
from company.invoice import Invoice

from company.person import *

from company import *
```

__init__.py

The “from module import *” syntax is very common, but does not work out of the box! Its namespace has to be defined in __init__.py.

```
# ./company/__init__.py

__all__ = [‘person’, ‘invoice’]
```

```
# ./company/person/__init__.py

from person import Person

__all__ = [‘Person’]
```

```
from company.person import *
```

simula consulting

simula

setup.py

The setup.py file is necessary to be able to install the package

```
from setuptools import setup

setup(name='company',
  version='0.1',
  description='A company.',
  url='http://url-to-git-repo',
  author='Developer',
  author_email='dev@example.com',
  license='MIT',
  packages=['company'],
  zip_safe=False)
```

```
pip install [-e] .
```

A note on installing packages using setup.py: It used to be common to install packages using

```
python setup.py install
```

Do not do this ever, especially when you don't specifically know the contents of the setup.py. Since it is just run like a normal Python script it can act like a Troyan horse for malicious code.

Specifying dependencies

Assume that our new package depends on both numpy and pandas, as well as a package that does not live in the PyPI space

```
from setuptools import setup

setup(...
  install_requires=[
    'numpy',
    'pandas'
  ],
  dependency_links=[
    'http://github.com/user/repo/tarball/master#egg=package-1.0'
  ]
  ...)
```

simula consulting

simula

Publication ready packages

A fairly complete guide on packaging and everything you might want to add to setup.py can be found here:
https://python-packaging.readthedocs.io/en/latest/index.html

If you plan to publish your code in any form, *at the minimum you should add unit tests and documentation* - these are outside of the scope of this course, but here are some resources that should help you:

- Unit testing using Pytest: https://docs.pytest.org/en/latest/index.html
- Documentation using Sphinx and Readthedocs: https://samnicholls.net/2016/06/15/how-to-sphinx-readthedocs/#fnref-841-4