ETH zürich

Introduction to Scientific Computation
Lecture 6
Fall 2021

OOP, Python modules, C/C++ in python

Mikhail Usvyatsov November 10, 2021

OOP

is a programming **paradigm.**

- Imperative programming
  - Object-oriented
  - Procedural
- Declarative programming
  - Functional
  - Logic

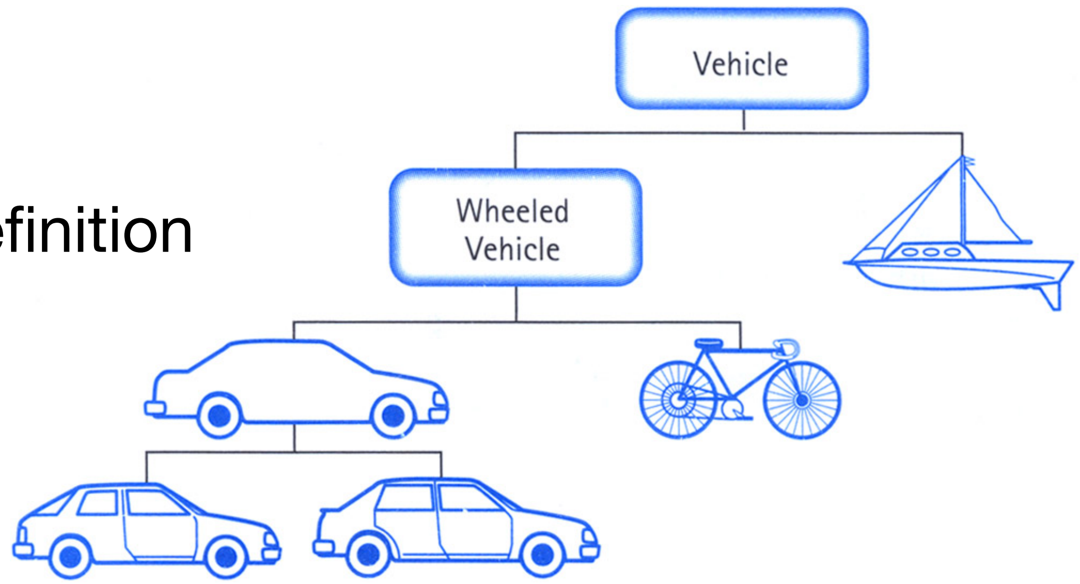
OOP

is good:

- **DRY** do not repeat yourself
- **KIS(S)** keep is simple

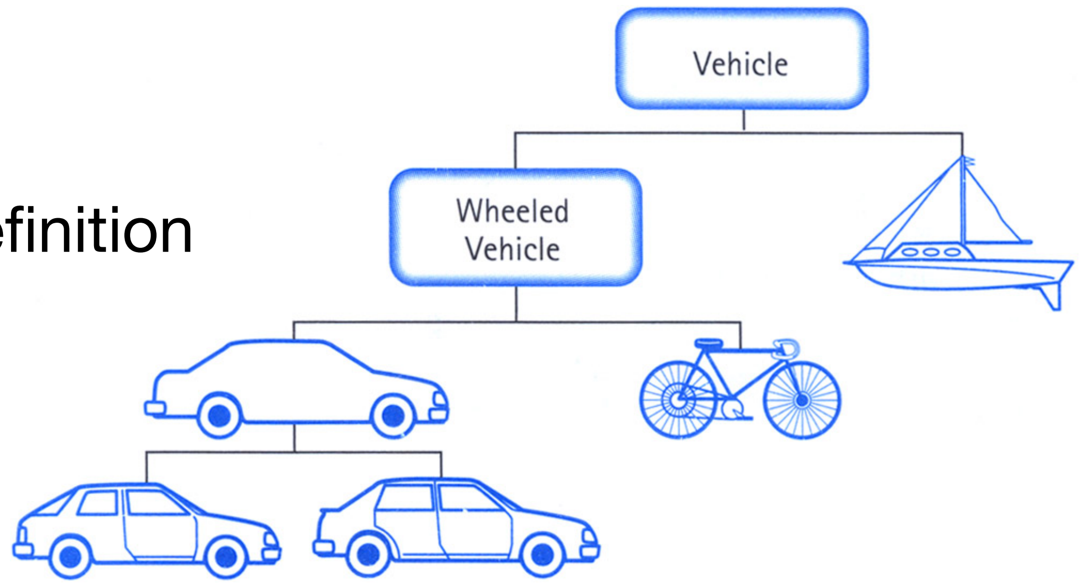
OOP

- Everything is an object
- Object properties are defined by its class definition
- Relations! Relations! Relations! matter



OOP

- Everything is an object
- Object properties are defined by its class definition
- Relations! Relations! Relations! matter

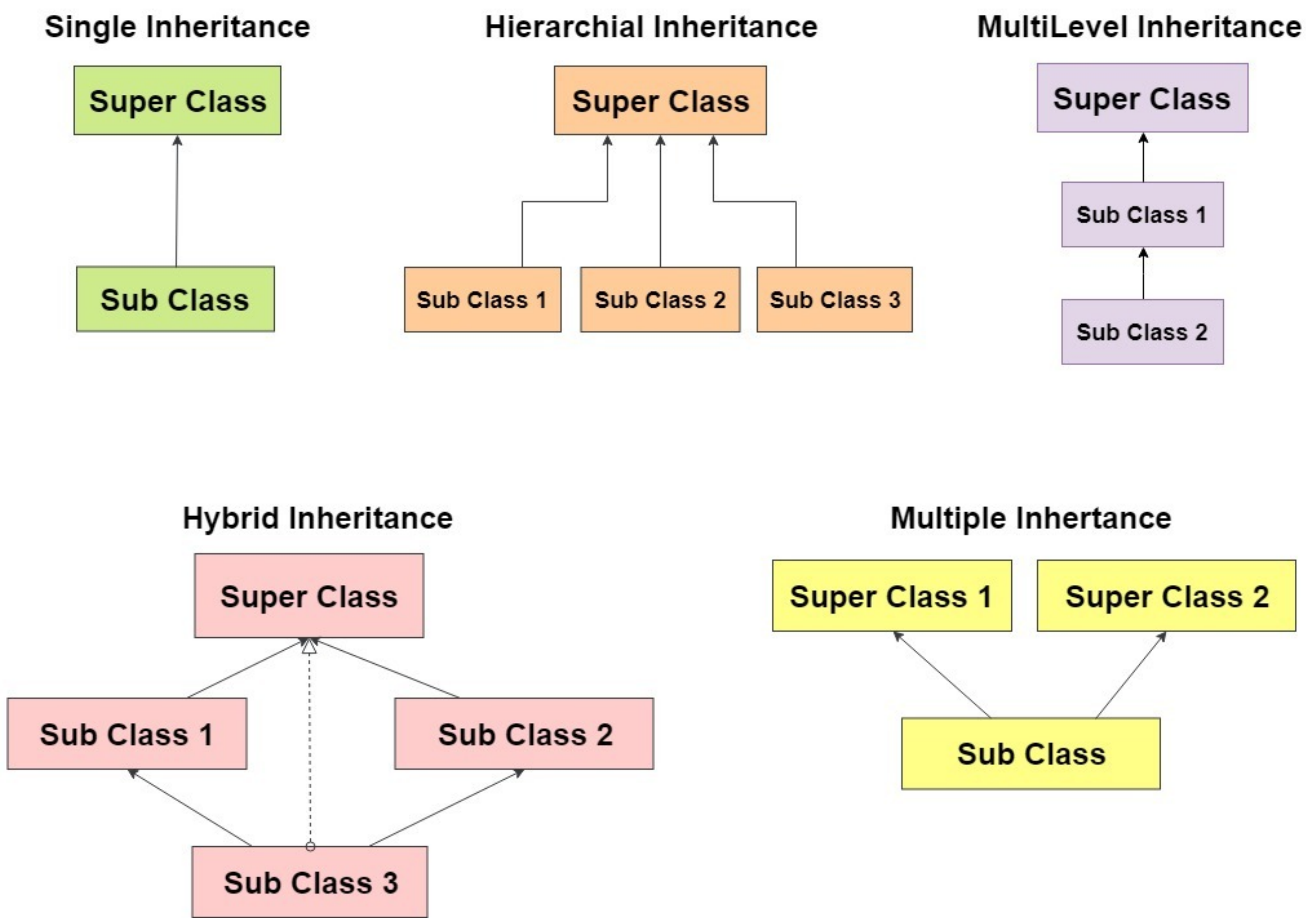


3 main principles

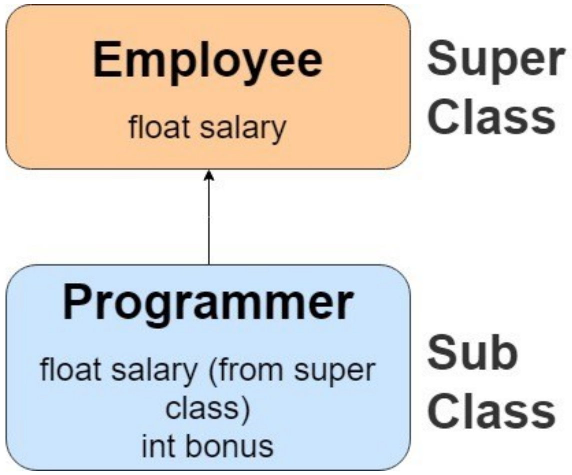
- Inheritance
- Encapsulation
- Polymorphism

Inheritance

- Possibility to define new classes based on existing



Inheritance

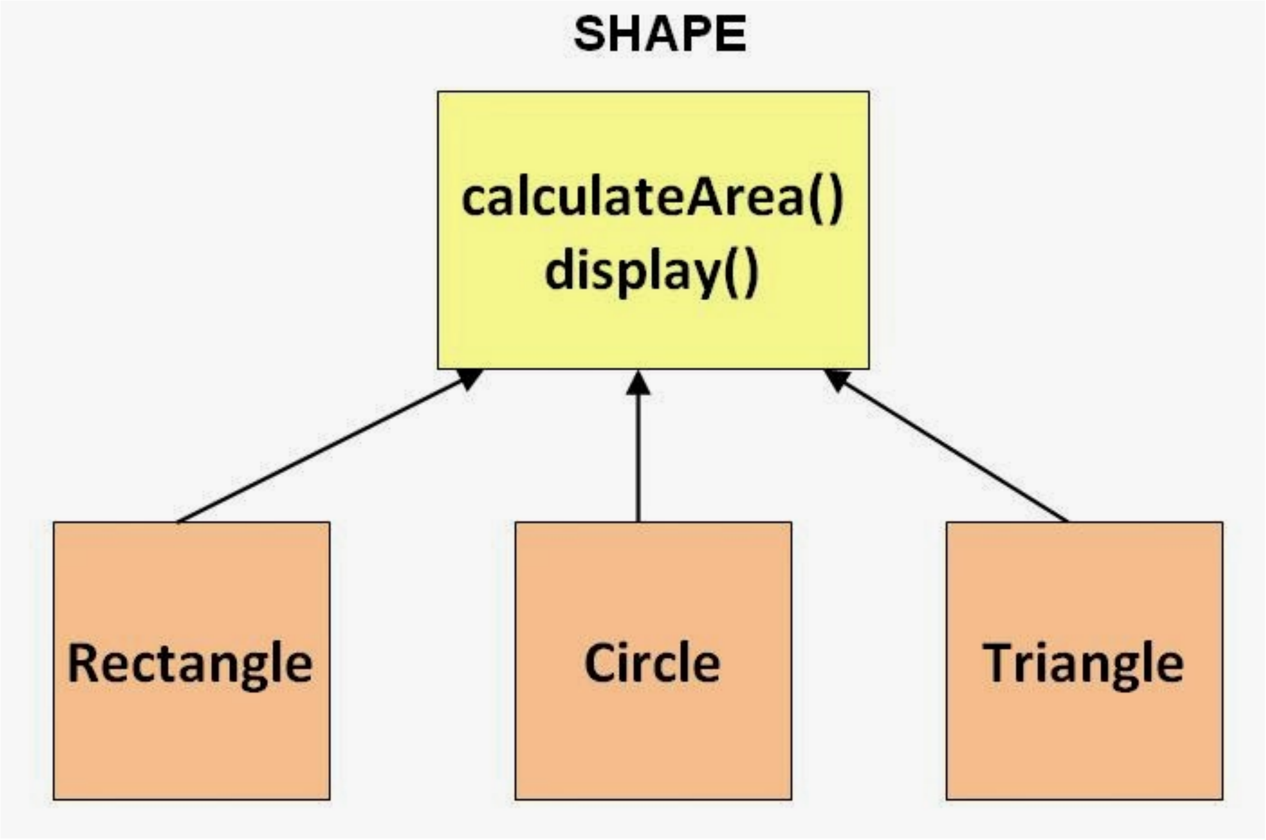


Encapsulation

- All object properties are stored privately
- There should exist methods for accessing the properties
- Implementation details are hidden to provide abstraction
- Abstraction should not leak the implementation details

Polymorphism

- Different classes might (re)-implement abstract inherited method on their own



Advanced OOP

- Multiple inheritance
- Monkey patching
- Abstract base classes
- Metaclasses

- In the most cases you do not need these… (KISS)

Multiple inheritance: method resolution order

C3 Linearization Algorithm
enforces following 2 constraints

- Children precede their parents
- If a class inherits from multiple classes, they are kept in the order specified in the tuple of the base class.

Also known as C3 super-class linearization, it is based on 3 rules

- Consistent extended precedence graph, which in short means how base class is extended from the super class. Inheritance graph determines the structure of method resolution order.
- Preserving local precedence ordering, i.e., visiting the super class only after the method of the local classes are visited.
- Monotonicity

In [12]:
```
O = object

class A(O): pass

class B(O): pass

class C(A,B) : pass

class D(B,A): pass

class E(C,D): pass
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-aaa942304035> in <module>
     15
     16
---> 17 class E(C,D): pass

TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, B
```

Multiple inheritance: method resolution order

```
1. C1 C2 C3...CN are the elements of the list of classes [C1, C2, C3..CN]
2. Head of the list is the first element C1
3. Tail of the list is the rest of the list C2...CN
4. The sum of the lists [C] + [C1, C2...CN] = C + (C1 C2....CN) = C C1 C2....CN
```

the linearization of C is the sum of C + the merge of the linearizations of the parents and list of parents.

```
L[C(B1...BN)]= C + merge(L[B1] L[B2].....L[BN])
L[object] = object
```

Merge:

- ***take the head of the first list, i.e L[B1][0];***
- ***if this head is not in the tail of any of the other lists, then add it to the linearization of C and remove it from the lists in the merge, otherwise look at the head of the next list and take it, if it is a good head.***
- ***Then repeat the operation until all the class are removed or it is impossible to find good heads.***

```
L[B(A)] = B + merge(L[A],A)
L[B(A)] = B + L[A]
L[B(A)] = B + A + L[object]
```

Monkey patching

```
# monk.py
class A:
     def func(self):
          print "func() is being called"
```

```
import monk
def monkey_f(self):
     print "monkey_f() is being called"

# replacing address of "func" with "monkey_f"
monk.A.func = monkey_f
obj = monk.A()

# calling function "func" whose address got replaced
# with function "monkey_f()"
obj.func()
```

Abstract class

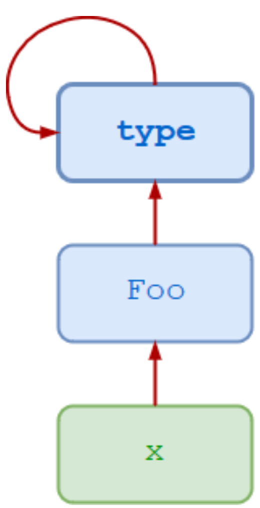
- Abstract class is a class with at least one abstract method
- Abstract class cannot be instantiated
- Classes inheriting from Abstract class must implement all its abstract methods

- See abc module in python

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    @abstractmethod
    def do_something(self):
        print("Some implementation!")

class AnotherSubclass(AbstractClassExample):
    def do_something(self):
        super().do_something()
        print("The enrichment from AnotherSubclass")

x = AnotherSubclass()
x.do_something()
```

Metaclass

- Class of class class is class class …



ETH zürich

Pattern

Design Patterns are typical solutions to commonly occurring problems in software **design**. They are blueprints that can be taken and customised to solve a particular **design** problem in your code.

Decorator

A decorator is the name used for a software design pattern. Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated.

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_whee():
    print("Whee!")
```

ETH zürich

Modules

- Each .py file in python is a module
- It contains Python statements and definitions
- __name__ is the global variable with might be used inside the module to find out its name
- sys.path contains search directories for modules
- It is initialised with the current directory and PYTHON_PATH
- __init__.py file is required to make python threat directory as containing modules

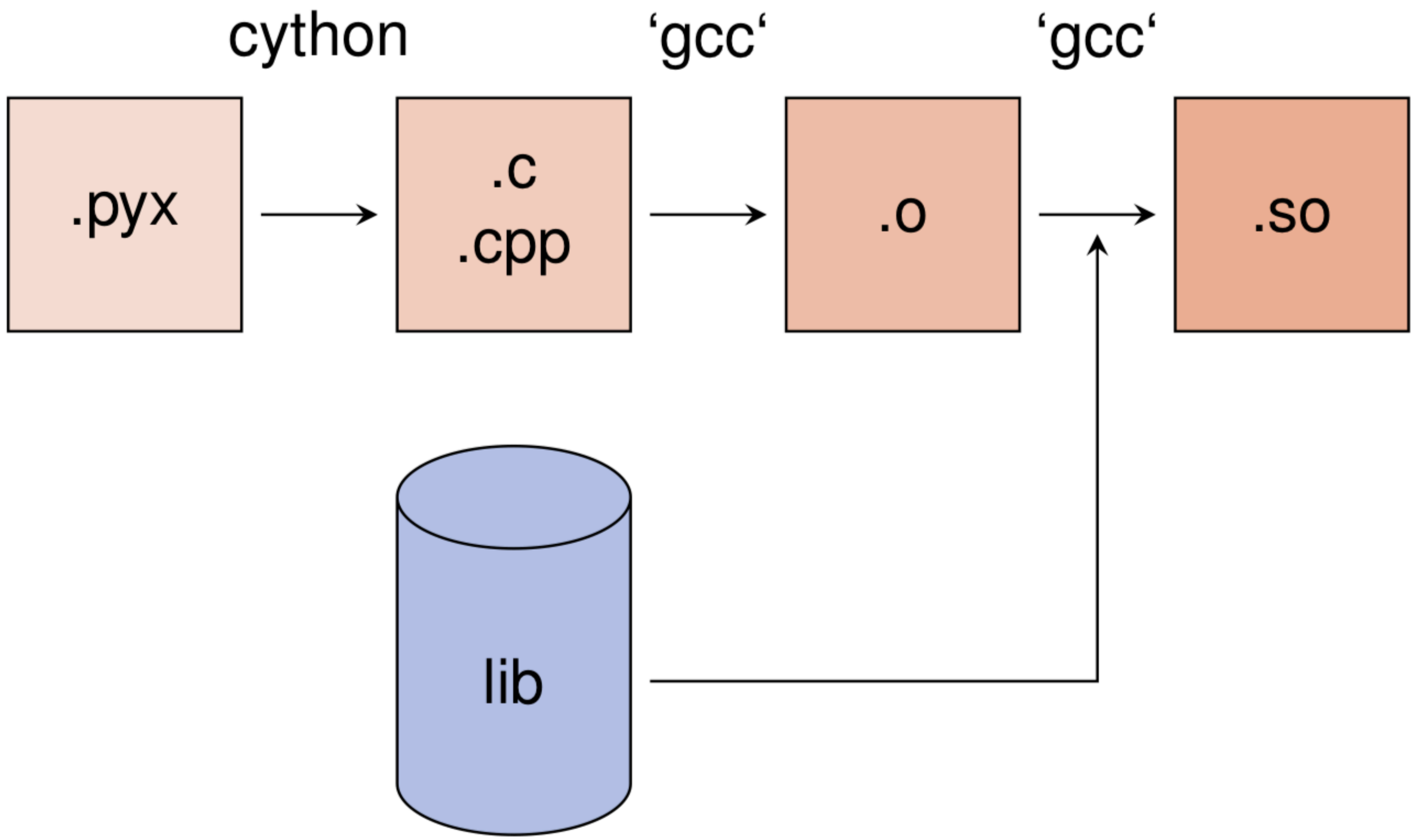
ETH zürich

C/C++ bindings

- Cython
- SWIG
- Pybind11

Cython

- A hybrid programming language/compiler
- Python statements are valid
- You have to provide argument types



SWIG

- Simplified Wrapper and Interface Generator
- You have to define interface files, the rest SWIG will do for you