Class hierarcies

- inheritance
- method overriding
- super
- multiple inheritance

Calling methods of a class

- If an object *obj* of class *C* has a method `method`, then usually you call *obj*`.`*method*`()`
- It is possible to call the method in the class directly using *C*`.`*method*, where the object is the first argument

  C`.`*method*`(`*obj*`)`

X.py

```
class X:
    def set_x(self, x):
        self.x = x

    def get_x(self):
        return self.x

obj = X()

obj.set_x(42)

print(f'{obj.get_x() = }')
print(f'{obj.x = }')
print(f'{X.get_x(obj) = }')
```

Python shell

```
| obj.get_x() = 42
| obj.x = 42
| X.get_x(obj) = 42
```

Classes and Objects

Observation: students and employees are persons with additional attributes

```
class Person
set_name(name)
get_name()

set_address(address)
get_address()
```

instance

```
Person object
name = 'Mickey Mouse'
address = 'Mouse Street 42, Duckburg'
```

```
class Student
set_name(name)
get_name()

set_address(address)
get_address()

set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

instance

```
Student object
name = 'Donald Duck'
address = 'Duck Steet 13, Duckburg'
id = '1094'
grades = {'programming' : 'A' }
```

```
Employee object
name = 'Goofy'
address = 'Clumsy Road 7, Duckburg'
employer = 'Yarvard University'
```

Classes and Objects

```
class Person
set_name(name)
get_name()

set_address(address)
get_address()
```

```
class Student
set_name(name)
get_name()

set_address(address)
get_address()

set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

person attributes

Goal – avoid redefining the 4 methods below from `person` class again in `student` class

person.py

```
class Person:
    def set_name(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_address(self, address):
        self.address = address

    def get_address(self):
        return self.address
```

Classes inheritance

```
class Person
```

```
set_name(name)
get_name()

set_address(address)
get_address()
```

```
class Student
```

```
set_name(name)
get_name()

set_address(address)
get_address()
```

person attributes

```
set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

class `Student` inherits from class `Person`
class `Person` is the base class of `Student`

person.py

```
class Student(Person):
    def set_id(self, student_id):
        self.id = student_id

    def get_id(self):
        return self.id

    def set_grade(self, course, grade):
        self.grades[course] = grade

    def get_grades(self):
        return self.grades
```

Classes constructors

```
class Person
set_name(name)
get_name()

set_address(address)
get_address()
```

```
class Student
set_name(name)
get_name()

set_address(address)
get_address()

set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

person attributes

person.py

```
class Person:
    def __init__(self):
        self.name = None
        self.address = None
    ...

class Student(Person):
    def __init__(self):
        self.id = None
        self.grades = {}
        Person.__init__(self)
    ...
```

constructor for `Person` class

constructor for `Student` class

Notes

1) If `Student.__init__` is not defined, then `Person.__init__` will be called
2) `Student.__init__` must call `Person.__init__` to initialize the `name` and `address` attributes

super()

class Person

```
set_name(name)
get_name()

set_address(address)
get_address()
```

class Student

```
set_name(name)
get_name()

set_address(address)
get_address()
```

person attributes

```
set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

person.py

```
class Person:
    def __init__(self):
        self.name = None
        self.address = None
    ...

class Student(Person):
    def __init__(self):
        self.id = None
        self.grades = {}
        Person.__init__(self)
        super().__init__()
    ...
```

alternative constructor

Notes

1) Function `super()` searches for attributes in base class
2) `super` is often a keyword in other OO languages, like Java and C++
3) Note `super().__init__()` does not need `self` as argument

Method search order

```
class Person
```

```
set_name(name)
get_name()

set_address(address)
get_address()
```

parent class

```
class Student(Person)
```

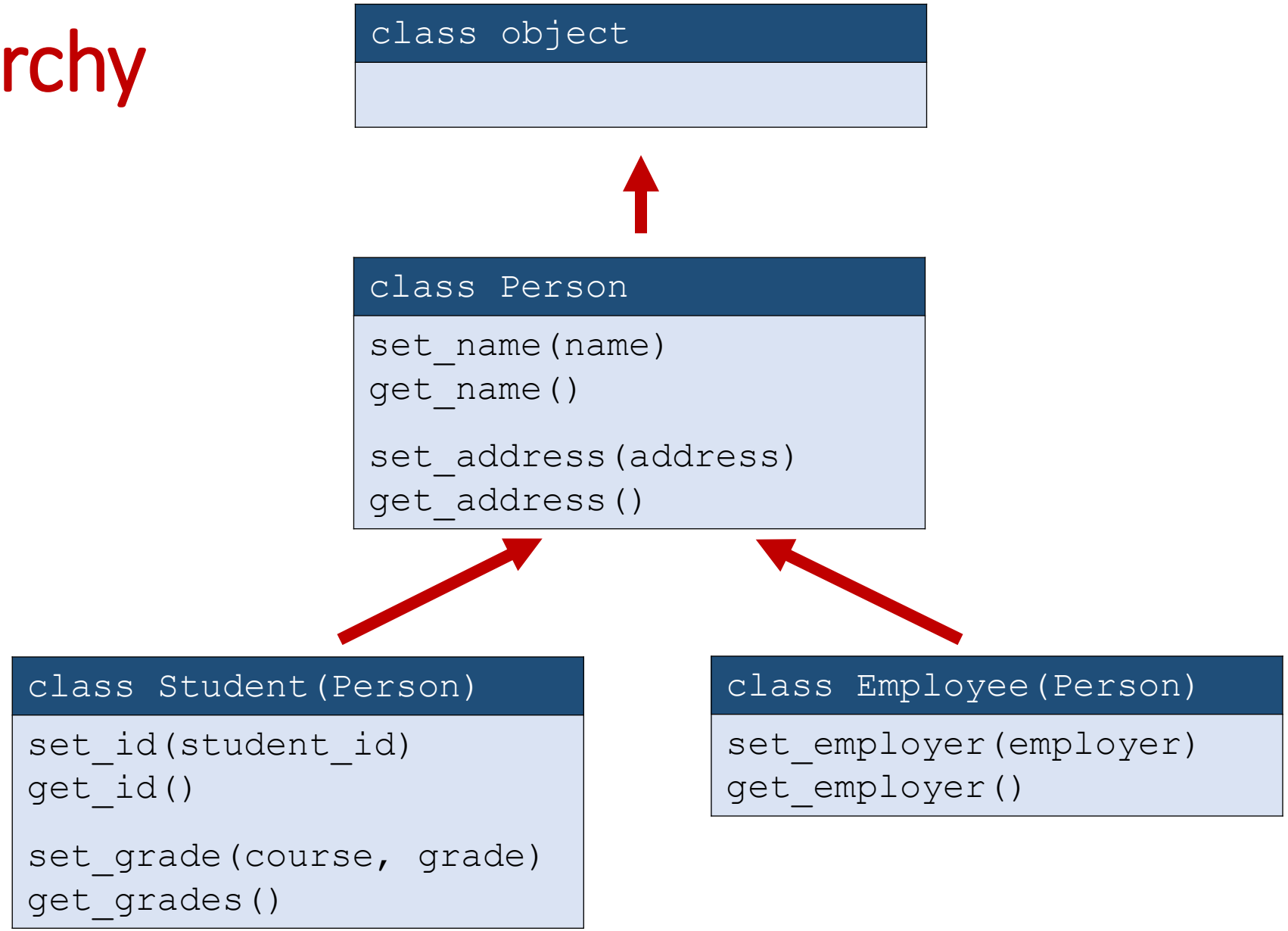
```
set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

instance of

```
Student object
```

```
name = 'Donald Duck'
address = 'Duck Steet 13, Duckburg'
id = '1094'
grades = {'programming' : 'A' }
```

Class hierarchy



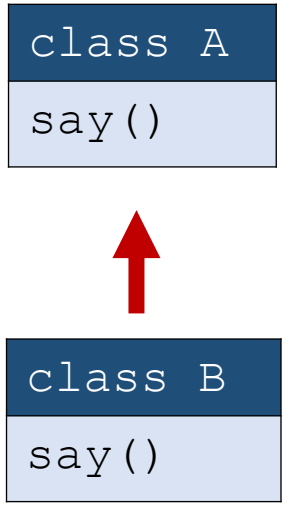
Method overriding

overloading.py

```
class A:
    def say(self):
        print('A says hello')

class B(A):  # B is a subclass of A
    def say(self):
        print('B says hello')
        super().say()
```

Python shell

```
> B().say()
| B says hello
| A says hello
```



In Java one can use the keyword "`finally`" to prevent any subclass to override a method

Question – What does `b.f()` print ?

Python shell

```
> class A:
      def f(self):
          print("Af")
          self.g()
      def g(self):
          print("Ag")

> class B(A):
      def g(self):
          print("Bg")

> b = B()
> b.f()
| ?
```

a) `AttributeError`

b) `Af Ag`

c) `Af Bg`

d) Don’t know

Undefind methods in superclass ?

Python shell

```
> class A:
      def f(self):
          print("Af")
          self.g()
      def g(self):
          print("Ag")
> class B(A):
      def g(self):
          print("Bg")
> b = B()
> b.f()
| Af
| Bg
> a = A()
> a.f()
| Af
| Ag
```

Python shell

```
> class A:
      def f(self):
          print("Af")
          self.g()
> class B(A):
      def g(self):
          print("Bg")
> b = B()
> b.f()
| Af
| Bg
> a = A()
> a.f()
| Af
| AttributeError: 'A' object has no attribute 'g'
```

method `g` undefined in class `A`; subclasses must implement `g` to be able to call `f`

in Java, `A` would have been required to be declared an *abstract class*

can create instance of `A`

fails since `g` is not defined in class `A`

Name mangling and inheritance 

Python shell

```
> class A:
      def f(self):
          print("Af")
          self.__g()
      def __g(self):
          print("Ag")
> class B(A):
      def __g(self):
          print("Bg")
> b = B()
> b.f()
| Af
| Ag
```

- The call to `A.__g` in `A.f` forces a call to `__g` to stay within `A`
- Recall that due to name mangling, `__g` is accessible as `A._A__g`

Multiple inheritance

- A class can inherit attributes from multiple classes (in example two)
- When calling a method defined in several ancestor classes, Python executes only one of the these (in the example `say_hello`)
- Which one is determined by the so called ”C3 Method Resolution Order” (originating from the Dylan language)

Raymond Hettinger, *Super considered super!*
Conference talk at PyCon 2015

multiple_inheritance.py

```
class Alice:
    def say_hello(self):
        print("Alice says hello")
    def say_good_night(self):
        print("Alice says good night")
class Bob:
    def say_hello(self):
        print("Bob says hello")
    def say_good_morning(self):
        print("Bob says good morning")
class X(Alice, Bob):  # Multiple inheritance
    def say(self):
        self.say_good_morning()
        self.say_hello()        # C3 resolution
        Alice.say_hello(self)   # from Alice
        Bob.say_hello(self)     # from Bob
        self.say_good_night()
```

Python shell

```
> X().say()
| Bob says good morning
| Alice says hello
| Alice says hello
| Bob says hello
| Alice says good night
```

since `Alice` before `Bob` in list of super classes

C3 Method resolution order

- Use `help(`*`class`*`)` to determine the resolution order for the class
- or access the `__mro__` attribute of the class

Python shell

```
> X.__mro__
| (<class '__main__.X'>, <class '__main__.Alice'>,
  <class '__main__.Bob'>, <class 'object'>)
> help(X)
| Help on class X in module __main__:
| class X(Alice, Bob)
|  |  Method resolution order:
|  |      X
|  |      Alice
|  |      Bob
|  |      builtins.object
|  |  Methods defined here:
|  |  say(self)
|  |  ----------------------------------
|  |  Methods inherited from Alice:
|  |  say_good_night(self)
|  |  say_hello(self)
|  |  ----------------------------------
|  |  ...
|  |  ----------------------------------
|  |  Methods inherited from Bob:
|  |  say_good_morning(self)
```

Question – Who says hello ? Bob says good morning

inheritance.py

```
class Alice:
    def say_hello(self):
        print("Alice says hello")
class Bob:
    def say_hello(self):
        print("Bob says hello")
    def say_good_morning(self):
        self.say_hello()
        print("Bob says good morning")
class X(Alice, Bob):  # Multiple inheritance
    pass

X().say_good_morning()
```

a) Alice

b) Bob

c) Dont' know

…example of code injection using multiple inheritance and where body of new class is empty

Comparing objects and classes

- `id(obj)` returns a unique identifyer for an object (in CPython the memory address)
- `obj1 is obj2` tests if `id(obj1) == id(obj2)`
- `type(obj)` and `obj.__class__` return the class of an object
- `isinstance(object, class)` checks if an object is of a particular class, *or a derived subclass*
- `issubclass(class1, class2)` checks if `class1` is a subclass of `class2`

Note: PEP8 recommends to use `isinstance(x, int)` over `type(x) is int`

`is` is not for integers, strings, ... and `is` is not `==`

Python shell

```
> 500 + 500 is 1000
| True
> x = 500
> x + x is 1000
| False
> x + x == 1000  # int.__eq__(...)
| True
> for x in range(0, 1000):
      if x - 1 + 1 is not x:
          print(x)
          break
| 257
> for x in range(0, -1000, -1):
      if x + 1 - 1 is not x:
          print(x)
          break
| -6
```

Python shell

```
> "abc" is "abc"
| True
> "abc" is "xabc"[1:]
| False
> x, y = "abc", "xabc"[1:]
> x, y
| ('abc', 'abc')
> x is y
| False
> x == y  # x.__eq__(y)
| True
```

- Only use `is` on objects !
- Even though `isinstance(42, object)` and `isinstance("abc", object)` are true, do not use `is` on integers and strings !

Comparison of OO in Python, Java and C++

- private, public, ... – in Python everything in an object is public
- class inheritance – core concept in OO programming
  - Python and C++ support multiple inheritance
  - Java only allows single inheritance, but Java ”interfaces” allow for something like multiple inheritance
- Python and C++ allow overloading standard operators (+, *, ...).
  In Java it is not possible
- Overloading methods
  - Python extremely dynamic (hard to say anything about the behaviour of a program in general)
  - Java and C++’s type systems allow several methods with same name in a class, where they are distinguished by the type of the arguments, whereas Python allows only one method that can have * and ** arguments

Python is really dynamic…
(this is ugly – likely don't do this at home)

Python shell

```
> class Pair:
      def __init__(self, x, y):
          self._x = x
          self._y = y

> point = Pair(3, 5)
> print(point)  # class Pair has no __str__ method, uses object.__str__
| <__main__.Pair object at 0x0000027571904B50>

> Pair.__str__ = lambda self: f'Pair({self._x}, {self._y})'

> print(point)
| Pair(3, 5)
```

dynamically add a method to an existing class (and all existing instances),
e.g. technique used by the class decorator `@functools.total_ordering`

C++ example

- Multiple methods with identical name (`print`)
- The types distinguish the different methods

printing.py

```
class MyClass:
    def print(self, value):
        if isinstance(value, int):
            print('An integer', value)
        elif isinstance(value, str):
            print('A string', value)

C = MyClass()
C.print(42)
C.print('abc')
```

printing.cpp

```
#include <iostream>
using namespace std;

class MyClass {
public:
  void print(int x) {
    cout << "An integer " << x << endl;
  };
  void print(string s) {
    cout << "A string " << s << endl;
  };
};

main() {
  MyClass C;
  C.print(42);
  C.print("abc");
}
```

Shell

```
An integer 42
A string abc
```