Programming in Python[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

Academic year 2020/21, II semester

Lecture XVI: Inheritance

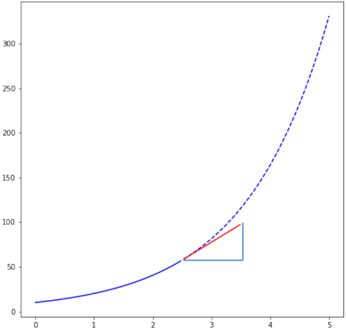
Destructuring a bound computation

```
def approx_euler(t: np.ndarray, f0: float, dfun:
↪  Callable[[float], float]) -> np.ndarray:
    """Compute the Euler approximation of a function on
    ↪  times t, with derivative dfun.
    """
    res = np.zeros_like(t)
    res[0] = f0

    for i in range(1, len(t)):
        res[i] = res[i-1] + (t[i]-t[i-1])*dfun(res[i-1])

    return res
```

Since we approximate the solution of a differential equation $p' = f(p, t)$, we used the trick of writing `dfun` as a function of p: this is why we call it by passing a point of `res` (and not of pyt). This trick makes it possible to compute it *together* with res itself (given the initial condition).

Two things together

A good way to keep two things separate (thus they can be changed independently), but together is the object-oriented approach: a `class` is a *small world* in which several computations are bound together, they share data and can depend one on each other.



OOP approach

```
class EulerSolver:
    """An EulerSolver object computes the Euler approximation of a differential equation
   ↪  p' = f(p, t).

    Create it by giving the f function, then set the initial condition P0.
    The approximate solution on a given time span is computed by the method solve.
    """

    def __init__(self, f: Callable[[float, float], float]):
        self.f = f

    def set_initial_condition(self, P0: float):
        self.P0 = P0

    def solve(self, time: np.ndarray) -> np.ndarray:
        """Compute p for t values over time."""
        self.t = time
        self.p = np.zeros_like(self.t)
        # ....

    def _diff(self, i: int) -> float:
        """Compute the differential increment at time of index i."""

        assert i >= 0
        # ...
```

How to use it

```
time = np.linspace(0, 5, 100)

solver = EulerSolver(lambda p, t: 0.7*p)
solver.set_initial_condition(10)
euler = solver.solve(time)
```

What we have gained

Conceptual steps are separated (but kept together by the class). We can decide to change one of them independently. Object-oriented programming has a feature to make this easy: inheritance

```
class RKSolver(EulerSolver):
    def _diff(self, i: int) -> float:
        """Compute the differential increment at time
        ↪  of index i."""

        assert i >= 0
        # use Runge-Kutta now!
        # overridden functionality is available with
        # super()._diff(i)
```

`RKSolver` inherits the methods of `EulerSolver` and it overrides the method `_diff`.

Substitution principle

If inheritance is done properly (unfortunately not trivial in many cases), the new class can be used wherever the old one was.

```
solver = RKSolver(lambda p, t: 0.7*p)
solver.set_initial_condition(10)
rk = solver.solve(time)
```

Overridden methods must be executable when the old ones were and their must produce at least the “same effects” (Liskov’s principle).