Intermediate Python -- Part 2

Thierry Géraud

2024


```
explicit_typing: bool = True  # do it, it saves lives!
```

Quick reminder

Test: write a Python program that input a number of lines and stores the strings in an appropriate structure

such as:

```
  *
***
```

or:

```
   *
  ***
*****
```

what was the main point of this exercise?

we want an algorithm (so a . . . . . .) such as:

```
input n         # numbner of lines
for each line
   print a series of spaces ' '  (how many?)
   print a series of stars '*'   (how many?)
an output has been created
```

print is for debugging purpose; afterwards, we will store...

a way to print---or store?---is:

```
from sys import stdout  # explain...

sys.stdout.write(char)  # "std::cout <<" in C++
                        # "System.out.print" in Java
```

```
a skeleton of code for the exercise here...
```

```
from sys import stdout

def pyramid_print(n):
    spc = 0    # first number of spaces, depending on 'n'
    lenx = 0   # first number of stars; first means on the 1st line
    for i in range(0, n):  # for each line
        for j in range(0, spc):
            sys.stdout.write(' ')  # start with space(s)
        for j in range(0, lenx):
            sys.stdout.write('*')  # then write star(s)
        sys.stdout.write('\n')
        # changes for the next line:
        spc -= 0
        lenx += 0  # to deal with the next iteration
    sys.stdout.flush()  # flush the *buffer* stdout

pyramid_print(3)  # just a test
```

replace zeros!

now we want something like:

```
from FIXME import FIXME

def pyramid_string_list(n: int) -> List[str]:
    out: List[str] = []

    # cut

    return out

print(''.join(pyramid_string_list(3)))
```

spot the two major differences with the 1st version, and discuss them (pros/cons)

last we want the simplest code:

```
def pyramid_string(n: int) -> str:
    assert n > 0  # run-time test (dev mode); no error handling (release)
    s: str = ''

    # cut

    return s
```

and finally an actual **program**:

```
# cut

if __name__ == '__main__':
    n = int(input(''))
    print(pyramid_string(n))
```

why is it a program, not just a script?

Another exercise (not mandatory, but highly recommanded if you feel not so self-confident with the 1st one): now the output is

```
*  *  *
*     *
*  *  *
```

or:

```
*  *  *  *  *
*           *
*           *
*           *
*  *  *  *  *
```

and the input 'n' shall be odd

your feedback: are you OK with all the notions involved in this exercise?

a program = data types and algorithms (functions)

Illustration:

```
# a data type

@dataclass            # a useful "decorator" to know
class University:
    name: str = None
    location: str = None

# an algorithm, dedicated on objects being a University

def relocalize(u: University):
    if u.location == "Kremlin-Bicetre":
        u.location = "Grand Paris"
```

With object-orientation:

- **a program = a set of types (and their relationships)**
- **a type = data + algorithms on these data**

Illustration:

- we want rooms and doors
  - thanks to the types Room and Door
  - note the use of the plural/singular -> explanation?
- a room has a number (data)
- a room can be equiped with a new door (algorithm)

in object-orientation, a class has *attributes* and *methods*:

description-level	**instance-level**	**meaning**
class	object	an entity (*)
attributes	data	its state
methods (*properties*)	routines	its behavior

the two following objects live independently:

```
d11 = Door(11)  # creates a new Door -> "constructs" a Door
d12 = Door(12)  # constructs another Door -> a new memory slot
d12.close()     # we want 'd12' to be closed
```

and they can be "part of" a "larger" object:

```
r1 = Room(number = 1)  # we have a new room
r1.doors.append(d11)   # a room have doors
r1.open()              # a room can be opened
# cut
for d in r1.doors:
    print(d.number)    # we can print this room door numbers
```

all these objects, some of them interacting with each other, make a program run / evolve

Excerpt from the previous slide:

```
# we have a new room
# a room have doors
# a room can be opened
   # we can print a room door number
```

what does that mean? what is the translation into Python?

`they can be "part of" a "larger" object`

is this programming?

Hint: if it is programming, then programming is . . . .

(discussion)

"a room have doors" means:

- any room have doors
- every rooms have doors
- the class 'Room' has an attribute 'doors'

"a room can be opened" means:

- any/every room/s can be opened
- the class 'Room' has a method 'open' (action *vs* reading data)

In

```
r1.doors
# cut
print(d.number)
```

we access some data of the room 'r1' and of the door 'd'

In

```
r1.open()
```

- we call an algorithm (piece of code / subroutine) **on** the room 'r1'
- we say that 'r1' is the *target* of this call / target = subject = recipient
- it reads: **"r1, do open"** -> execute this action (data can be modified) / this is an explicit message (we do not know the internals) / a program is **doing things**

```
r1.doors
# cut
print(d.number)
```

so we have

- 'doors' being a (public) attribute of the class 'Room'
- 'number' being a (public) attribute of the class 'Door'
- "(public)" means that we can directly access to data -> not a good idea, see later...

```
r1.open()
```

so

- 'open' is a (public) method of the class 'Room' = we can open a room
- we express that objects can do things / we can do things on objects
    - "object, do that" => my program evolves --- things are done
    - 'r1.open()' = **"r1, do open"** => we give/write instructions

a program is a set of instructions, but their are not linear, just like for a script

so

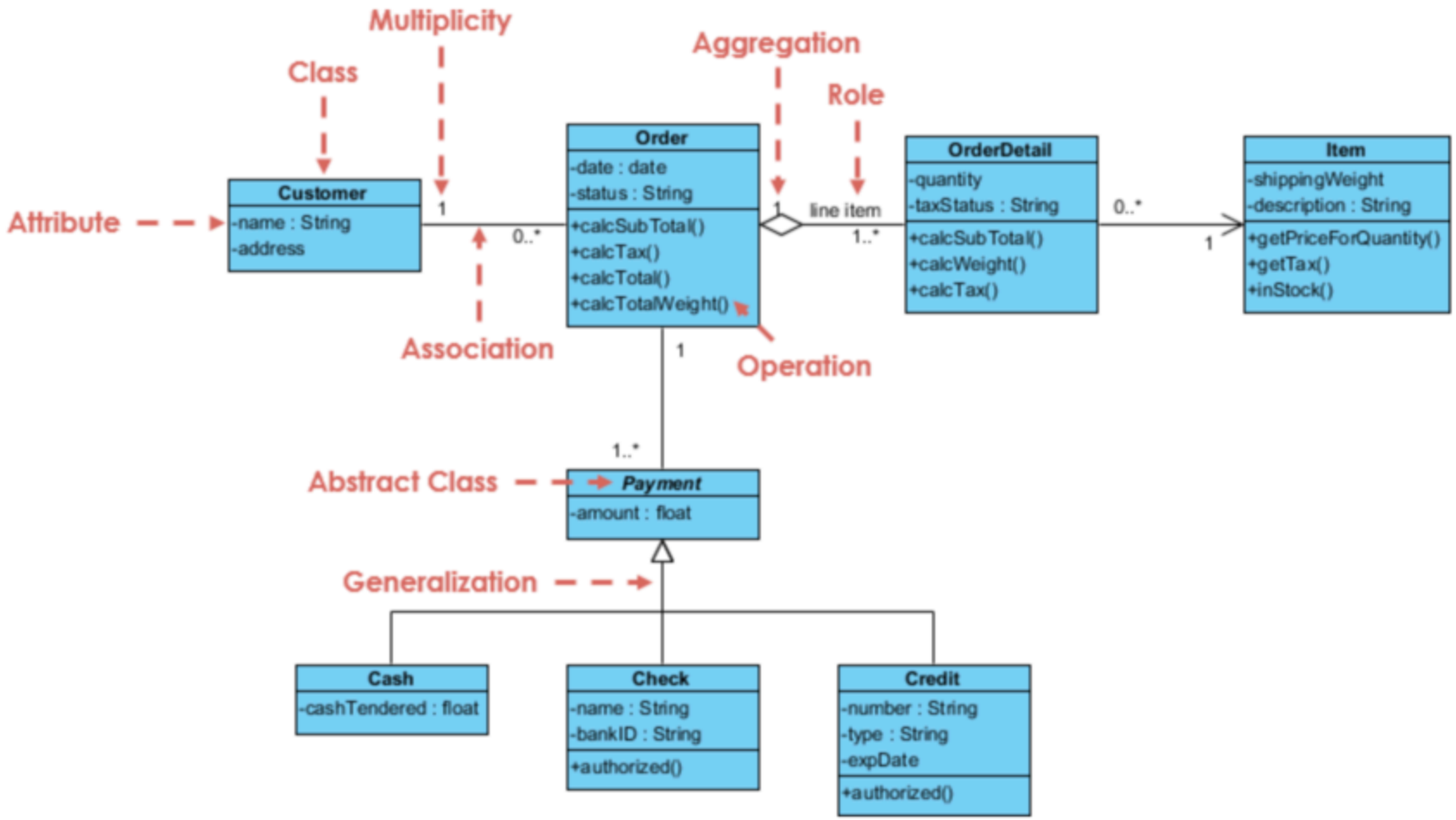
- the accessibility is *public*
  - meaning we can use it: access some data (attribute) / run an algorithm (method)
  - in the cas of methods, without knowing some internal details
    - that can be a very desirable property
  - note that accessing to raw data (attribute) and modify them can be risky
    - control what is done with data would be better...

What is Object-Orientation (OO)?

object-orientation features:

- objects / sometimes with classes
- data abstraction / encapsulation
- inclusion polymorphim / inheritance
- dynamic dispatch / message passing
- open recursion

an *object-oriented program* looks like:



(does that look like a script?)

Exercise: read the following class

```
class Door:
    """a very simple Door class"""

    def __init__(self, number: int, opened: bool = True):
        self.number = number
        self.opened = opened

    def open(self):
        self.opened = True
```

and this sample use:

```
dd = [Door(i) for i in range(10)]  # list comprehension
for d in dd:
    d.open()
```

Live step-by-step explanations:

```
FIXME
```

now we want to log when a door is opened so we write:

```
class Door:

    # cut

    def open(self):
        self.opened = True
        print(self, "is opened")

    def __str__(self):
        return "door " + str(self.number)
```

Explanations...

Reminders:

1. with object-orientation:
    - a program = a set of types (and their relationships)
    - a type = data + algorithms on these data
2. in object-orientation, a class has *attributes* and *methods*:

description-level	**instance-level**	**meaning**
class	object	an entity
attributes	data	its state
methods (*properties*)	(sub)routines	its behavior

Encapsulation in OO is: grouping attributes and methods

(we will see that it is also related with *information hidding*...)

Live exercice:

- let us define (right now) all together what can be a room
- so write a 'Room' class

Class hierarchies / Inheritance / inclusion polymorphism

this section is just about types are "organized" and about typing a set of types

dogs and cats *are* animals

- Animal is a superset of Dog -- we say that Animal is a superclass of Dog
- Cat is a subset of Animal -- we say that Cat is a subclass of Animal

(live drawing...)

the "is-a" relationship is called *generalization*

- Animal generalizes the notions of Cat's and Dog's

every animals can make sounds, so we define a method for that:

```
class Animal:
    def make_sound(self):
        pass   # meaning: do nothing
```

'self' (1st argument) designates the animal (object) that 'make_sound'

to call of this method:

```
a.make_sound()  # here 'self' is 'a' / actually not passed as an argument
```

'a' is not an argument in this method call; it is the *target*

so in:

```
class Animal:
    def make_sound(self):
        pass   # meaning: do nothing
```

'self' designates the target

```
class Dog(Animal):           # maps: a dog is an animal
    def make_sound(self):    # the method implementation for dogs
        print("bark")

class Cat(Animal):           # maps: a cat is an animal
    def make_sound(self):    # the method implementation for cats
        print("meow")

a = [Dog(), Cat()]  # a list of animals (dogs and cats are animals)
for e in a:
    e.make_sound()  # gives: bark meow
```

we have several implementations for 'make_sound' in different classes

when calling 'make_sound', a particular implementation is selected and run

```
a[0].make_sound()
a[1].make_sound()
```

'a[0]' is a dog (*) so the method 'make_sound' of 'Dog' is called

'a[1]' is a cat so the method 'make_sound' of 'Cat' is called

(*) meaning, an instance of the class Dog / an object with type Dog

```
aa: Animal = a[0]   # first element of our list of animals
print(id(aa) == id(a[0]))  # gives: True
```

we have:

- 'aa' designates 'a[0]'
  - it is a reference, not a copy
  - just like in

```
a: int = 1  # we expect an integer, and we get 1; correct
b = a  # 'b' is a reference to 'a'
```

- 'a[0]' is a 'Dog'
- 'aa' is a variable of type 'Animal'
- this initialization is: **we expect an animal, and we get a dog**
- so this is correct

```
aa.make_sound()     # a method is called on this object
```

we have:

- 'aa' is a variable of type 'Animal'
- we call the 'make_sound' method on this (instance of) 'Animal'
- yet we know that the **exact type** of 'aa' is actually 'Dog'
- the implementation of 'make_sound' that is called is the one of 'Dog'

static type: the one of the variable
dynamic type: the one of the object

when calling a method, the selected implementation is the one defined *in* the class of the target object...

...even though a superclass

now we want both attributes and methods:

- we keep the 'make_sound' method
- an attribute can be the animal name
- every data shall be initialized (good practice)
    - a special method is defined to construct objects
    - the constructor is named `__init__`

```python
class Animal:
    def make_sound(self):
        pass  # meaning: do nothing


class Dog(Animal):  # a dog is an animal

    def __init__(self, name: str = ""):  # a constructor (special method)
        self.name = name                 # (declares and) initializes an attribute

    def make_sound(self):   # a method
        print(self.name, "bark")  # access to the attribute 'name'

    def test(self):
        self.make_sound()  # just to illustrate "open recursion"
```

sample use:

```
d1 = Dog("Buddy")
d2 = Dog("Fluffy")

d1.make_sound()  # 'self' is 'd1' in this call
d2.make_sound()

print(d1.name == d2.name)  # gives: False
```

each object has its own attribute

we do likewise for Cat

we then have:

- make_sound for all animals
- a name for every dogs, and a name for every cats
    - that is redundant code (which is bad)

why not having a name for name for every animals

- so for dogs, cats, and upcoming new animal classes

let's equip the Animal class:

```
class Animal:

    def __init__(self, name: str = ""):  # constructor
        self.name = name                 # attribute

    def make_sound(self) -> None:        # polymorphic method
        pass

    def run(self) -> None:     # new method, available for all animals
        print("now running")
```

so we can now write:

```
d = Dog()
print(d.name)  # a dog has a name
d.run()        # a dog can run
```

we say that 'name' and 'run' are inherited (from the super class Animal)

```
class Animal:
    def __init__(self, name: str = ""):
        self.name = name                # attribute for each Animal
    # cut

class Dog(Animal):  # a dog is an animal

    def __init__(self, name: str = "", n_bones: int = 0):
        super().__init__(name)          # calls the Animal constructor
        self.n_bones: int = n_bones  # Dog has an extra attribute

    def make_sound(self) -> None:    # the method implementation for dogs
        print(self, "bark")
```

Dog has two attributes:

- 'name', inherited from the class Animal
- 'n_bones', declared in the class Dog

```
class Animal:
    # cut
    def run(self) -> None:
        print("now running")

class Dog(Animal):  # a dog is an animal

    def __init__(self, name: str = "", n_bones: int = 0):
        super().__init__(name)        # calls the Animal constructor
        self.n_bones: int = n_bones  # Dog has an extra attribute

    def make_sound(self) -> None:    # the method implementation for dogs
        print(self, "bark")
```

Dog has two methods:

- 'run', inherited from the class Animal
- 'make_sound', defined in the class Dog

plus its constructor

there is a difference between

- a "regular" attribute:
  - **one per object**
  - 'd1.name' and 'd2.name' are two distinct data
- a *class attribute* (or *class variable*)
  - **one for the class**
  - shared by all objects

```
class Animal:
    numbers: int = 0                # class attribute/variable
    def __init__(self, name: str = ""):
        self.name = name            # object attribute
        Animal.numbers += 1
    # cut
```

sample use:

```
d = [Dog(), Dog(), Cat()]
print(Animal.numbers)   # gives: 3
```