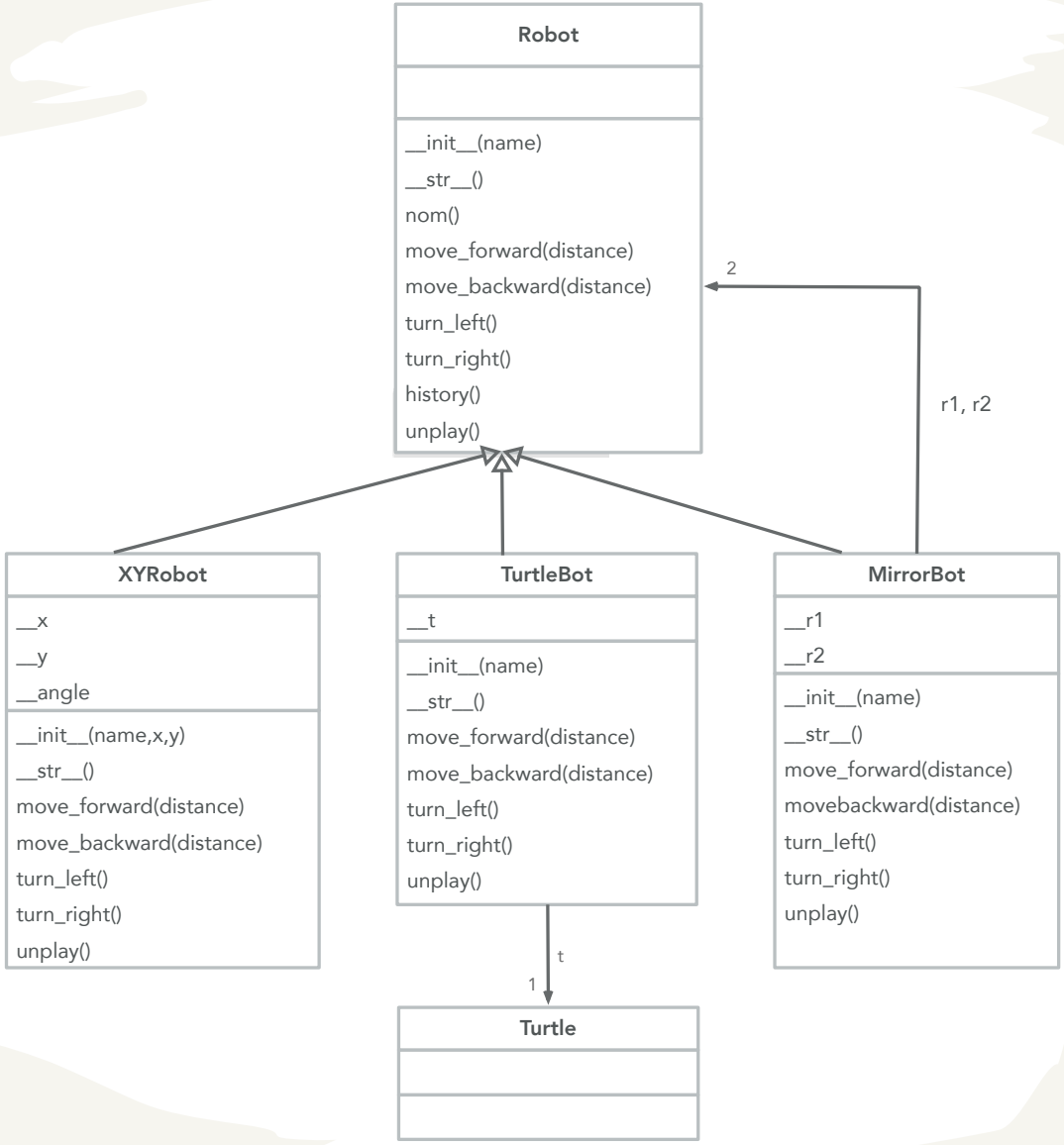
Informatique 1
Introduction à la programmation

Mission 10 : restructuration

Kim Mens Siegfried Nijssen Charles Pecheur


Robot
__init__(name)
__str__()
nom()
move_forward(distance)
move_backward(distance)
turn_left()
turn_right()
history()
unplay()
2
r1, r2
XYRobot
__x
__y
__angle
__init__(name,x,y)
__str__()
move_forward(distance)
move_backward(distance)
turn_left()
turn_right()
unplay()
TurtleBot
__t
__init__(name)
__str__()
move_forward(distance)
move_backward(distance)
turn_left()
turn_right()
unplay()
t
1
Turtle
MirrorBot
__r1
__r2
__init__(name)
__str__()
move_forward(distance)
movebackward(distance)
turn_left()
turn_right()
unplay()

Les tests unitaires (restructuration)

- Quelques points auxquels il faut faire attention :
  - Les tests unitaires sont important en Python à cause du typage dynamique
  - Chaque nom de méthode test commence par `test`
  - N'oubliez pas de
    - importer `unittest`
    - hériter de `unittest.TestCase`
    - mettre `self.` avant `assertEqual` etc.
  - Utilisation de la méthode `setup`
  - Différentes méthodes `assert*` existent

Implémentation : tests unitaires

```
import unittest
from TurtleBot import TurtleBot

class TestTurtleBot(unittest.TestCase):

    def setUp(self):
        self.t = TurtleBot("tBot")

    def test_init(self):
        self.assertEqual(self.t.getangle(), 0)
        self.assertEqual(self.t.position(), (0,0))

    def test_turnleft(self
        expectedposition = self.t.position()
        expectedangle = (self.t.getangle() + 90) % 360
        self.t.turnleft()
        self.assertAlmostEqual(self.t.getangle(),expectedangle)
        self.assertEqual(self.t.position(),expectedposition)
```

On devra hériter de la classe `TestCase` de cette librairie

la classe `TestCase` fournit des méthodes comme `setup`, `teardown`, `run` et les méthodes `assert*`

`setUp` est appelé avant chaque méthode de test

N'oubliez pas de mettre `self.` devant

Les méthodes de test commencent par le mot `test` pour informer la librairie quels tests à exécuter

Différentes méthodes assert* existent

Implémentation : tests unitaires

```
class TestTurtleBot(unittest.TestCase):
  ...
  def test_turnleft(self):
    expected_position = self.t.position()
    expected_angle = (self.t.getangle() + 90) % 360
    self.t.turnleft()
    self.assertAlmostEqual(self.t.getangle(), expected_angle, \
       msg = "Your turtleBot took a wrong turn or                \
       did not update its angle while turning left")
    self.assertEqual(self.t.position(), expected_position,     \
       msg = "Your turtleBot changed position while turning left")
  ...

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

0 ***(silencieux)*** : vous obtenez juste le nombre total de tests exécutés et le résultat global

1 ***(par défaut)*** : idem mode silencieux + un « . » pour chaque test réussi et un « F » pour chaque échec

2 ***(verbose)*** : vous obtenez le message d'aide de chaque test et le résultat

Les principales méthodes d'assertion

Méthode	Explications
`assertEqual(a,b)`	a et b ont la même valeur ou contenu
`assertIs(a,b)`	a et b sont ou pointent vers le même objet
`assertIsNot(a,b)`	a et b sont des objets différents
`assertIsNone(x)`	x est égal à None
`assertIsNotNone(x)`	x n'est pas égal à None
`assertIsInstance(a,b)`	a est une instance de type b
…	…

Les principales méthodes d'assertion

Méthode	Explications
`assertNotEqual(a,b)`	a != b
`assertAlmostEqual(a,b)`	a et b sont approximativement égaux
`assertTrue(x)`	x is True
`assertFalse(x)`	x is False
`assertIn(a,b)`	a in b
`assertNotIn(a,b)`	a not in b
`assertNotIsInstance(a,b)`	not isinstance(a, b)
`assertRaises(exception, fonction,*args,**kwargs)`	Vérifie que la fonction lève l'exception attendue.

https://docs.python.org/3/library/unittest.html#unittest.TestCase

Tester la classe Compte

```
class Compte :

    def __init__(self, titulaire):
        self.__titulaire = titulaire
        self.__solde = 0

    def titulaire(self):
        return self.__titulaire

    def solde(self):
        return self.__solde

    def __str__(self) :
        return "Compte de "+self.titulaire()+" : solde = "+str(self.solde())

    def deposer(self, somme):
        self.__solde += somme
        return self.solde()

    def retirer(self, somme):
        if self.solde() >= somme :
            self.__solde -= somme
            return self.solde()
        else :
            return "Solde insuffisant"
```

La classe `TestCompte` (v1.0)

```
import unittest

class TestCompte(unittest.TestCase):

    def setUp(self):
        self.transactions = [10,20,50,10]
        self.compte = Compte("test")

    def test_init(self):
        self.assertEqual(self.compte.solde(), 0, "Solde initial n'est pas zéro")

    def test_deposer(self):
        for somme in self.transactions :
            avant = self.compte.solde()
            self.compte.deposer(somme)
            apres = self.compte.solde()
            self.assertEqual(apres,avant+somme)

    def test_retirer(self):
        self.compte.deposer(50)
        for somme in self.transactions :
            avant = self.compte.solde()
            self.compte.retirer(somme)
            apres = self.compte.solde()
            self.assertEqual(apres,avant-somme)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

Exécuter la classe `TestCompte`

```
>>> %Run comptetest.py
test_deposer (__main__.TestCompte) ... ok
test_init (__main__.TestCompte) ... ok
test_retirer (__main__.TestCompte) ... FAIL

======================================================================
FAIL: test_retirer (__main__.TestCompte)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/kimmens/Cours/INFO1/code-theory-week-11/comptetest.py", line 55, in test_retirer
    self.assertEqual(apres,avant-somme)
AssertionError: 20 != -30

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

La classe `TestCompte` (v2.0)

```
import unittest

class TestCompte(unittest.TestCase):

    ...

    def test_retirer(self):
        self.compte.deposer(50)
        for somme in self.transactions :
            avant = self.compte.solde()
            res = self.compte.retirer(somme)
            apres = self.compte.solde()
            if somme > avant :
              self.assertEqual(apres,avant)
              self.assertEqual(res,"Solde insuffisant")
            else :
              self.assertEqual(apres,avant-somme)
              self.assertEqual(res,avant-somme)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

Exécuter la classe `TestCompte`

```
>>> %Run comptetest.py
test_deposer (__main__.TestCompte) ... ok
test_init (__main__.TestCompte) ... ok
test_retirer (__main__.TestCompte) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

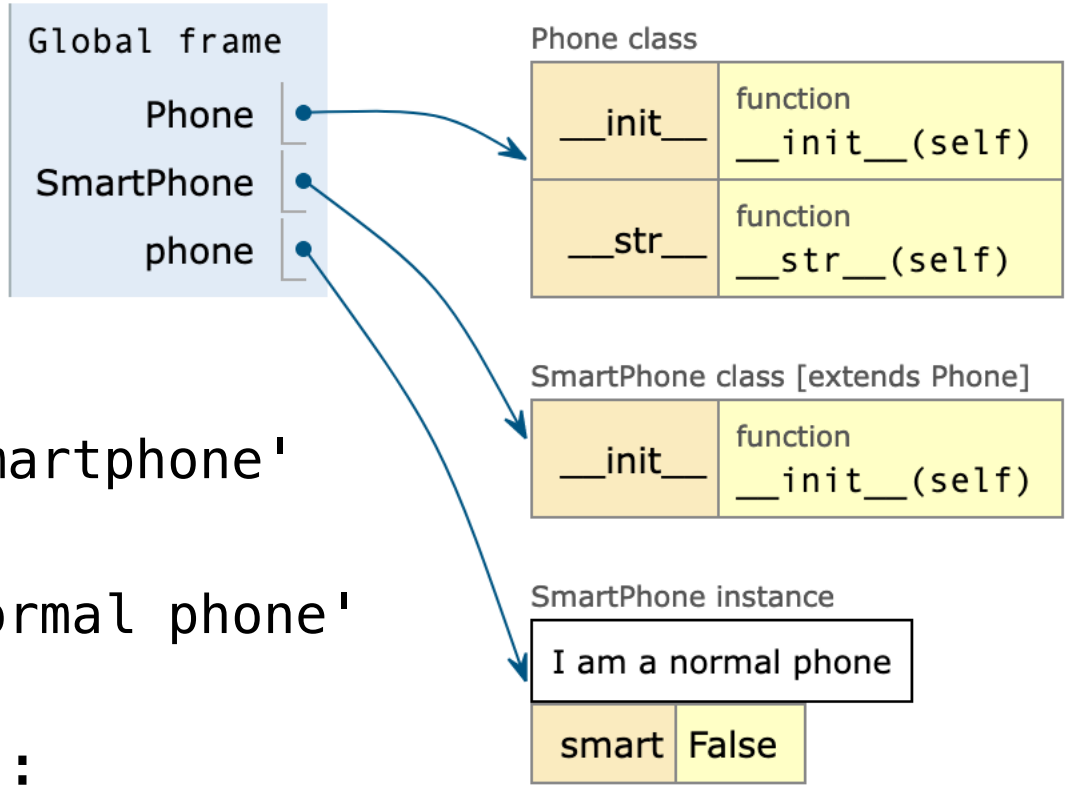
La portée des variables

- variable d'instance
  - publique
  - privée

La portée d'une variable d'instance publique

```
class Phone :
  def __init__(self):
    self.smart = False
  def __str__(self):
    if self.smart :
      return 'I am a smartphone'
    else :
      return 'I am a normal phone'

class SmartPhone(Phone):
  def __init__(self):
    self.smart = True
    super().__init__()
```

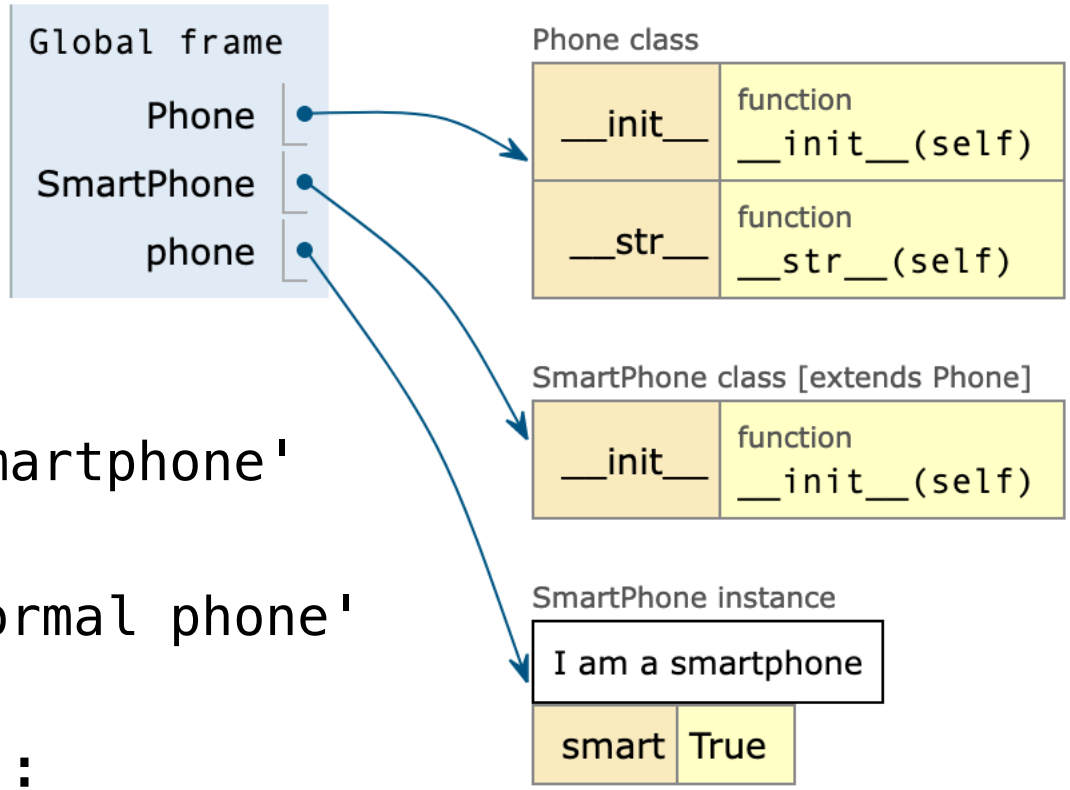



```
>>> phone = SmartPhone()
>>> print(phone)
I am a normal phone
```

La portée d'une variable d'instance publique

```
class Phone :
  def __init__(self):
    self.smart = False
  def __str__(self):
    if self.smart :
      return 'I am a smartphone'
    else :
      return 'I am a normal phone'

class SmartPhone(Phone):
  def __init__(self):
    super().__init__()
    self.smart = True
```

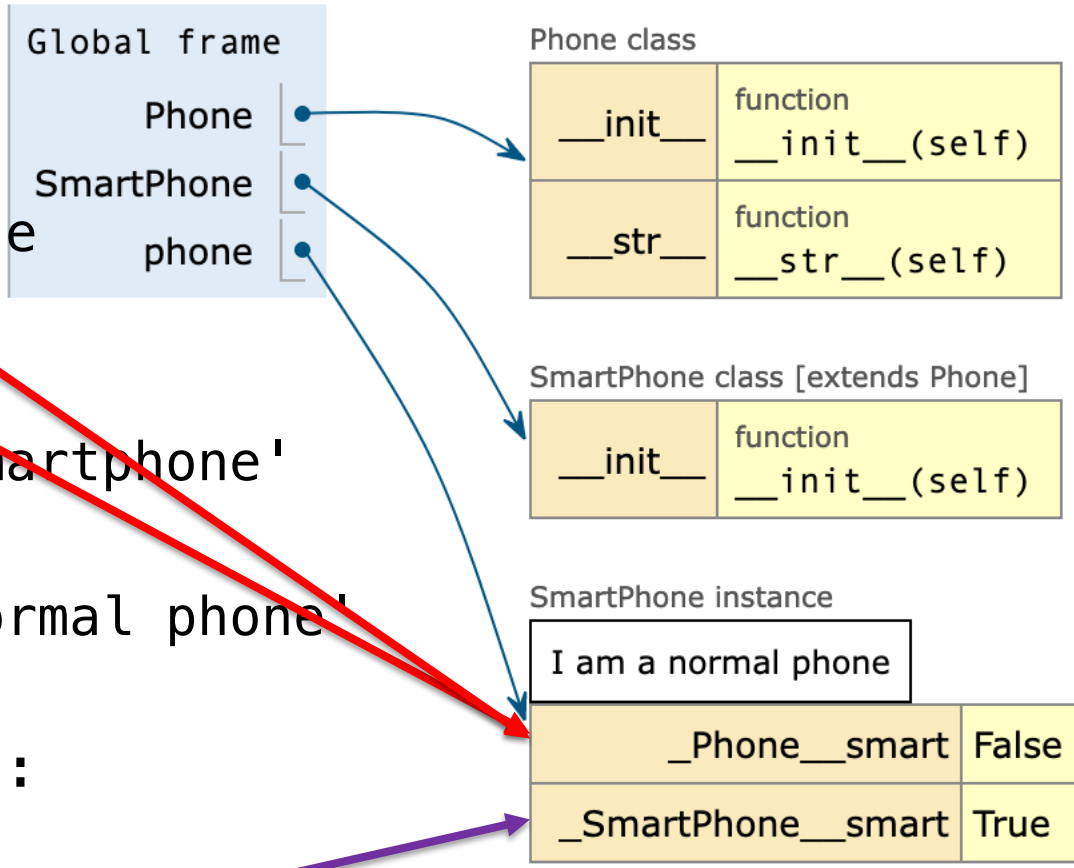



```
>>> phone = SmartPhone()
>>> print(phone)
I am a smartphone
```

La portée d'une variable d'instance privée

```
class Phone :
  def __init__(self):
    self.__smart = False
  def __str__(self):
    if self.__smart :
      return 'I am a smartphone'
    else :
      return 'I am a normal phone'

class SmartPhone(Phone):
  def __init__(self):
    super().__init__()
    self.__smart = True
```

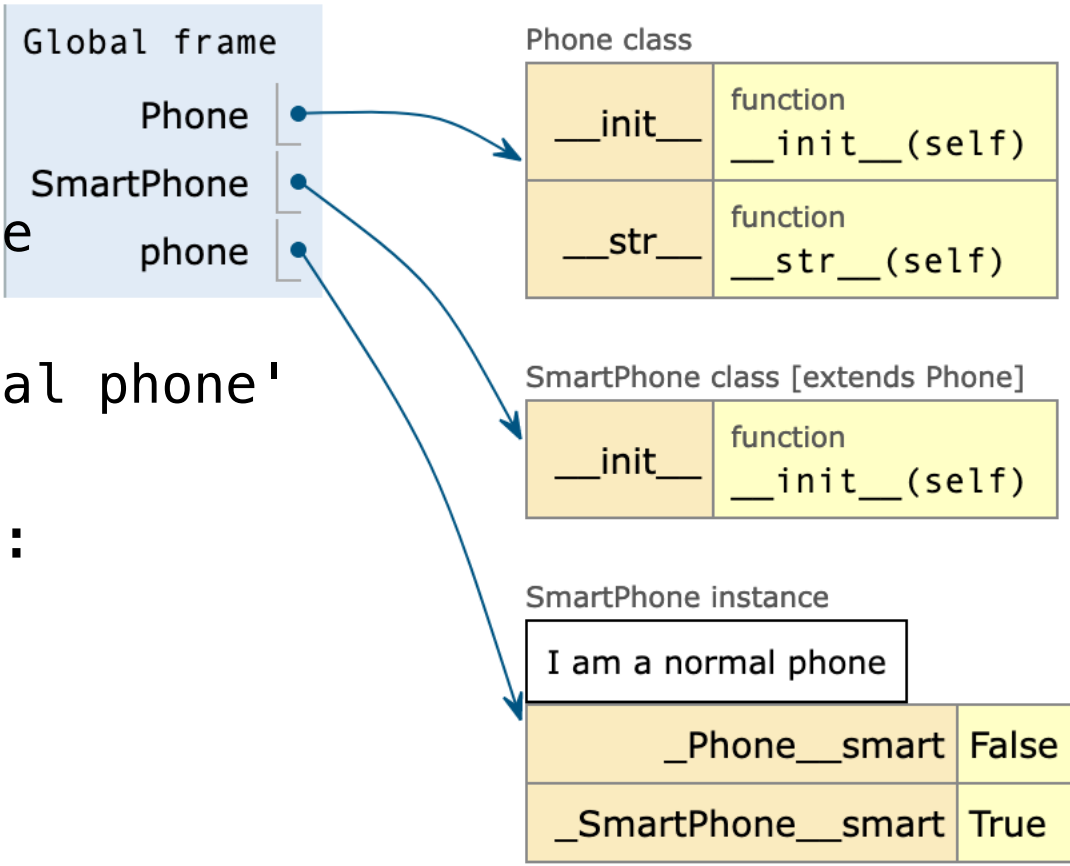



```
>>> phone = SmartPhone()
>>> print(phone)
I am a normal phone
```

La portée d'une variable d'instance privée

```
class Phone :
  def __init__(self):
    self.__smart = False
  def __str__(self):
    return 'I am a normal phone'

class SmartPhone(Phone):
  def __init__(self):
    super().__init__()
    self.__smart = True
  def __str__(self):
    return 'I am a smartphone'
```

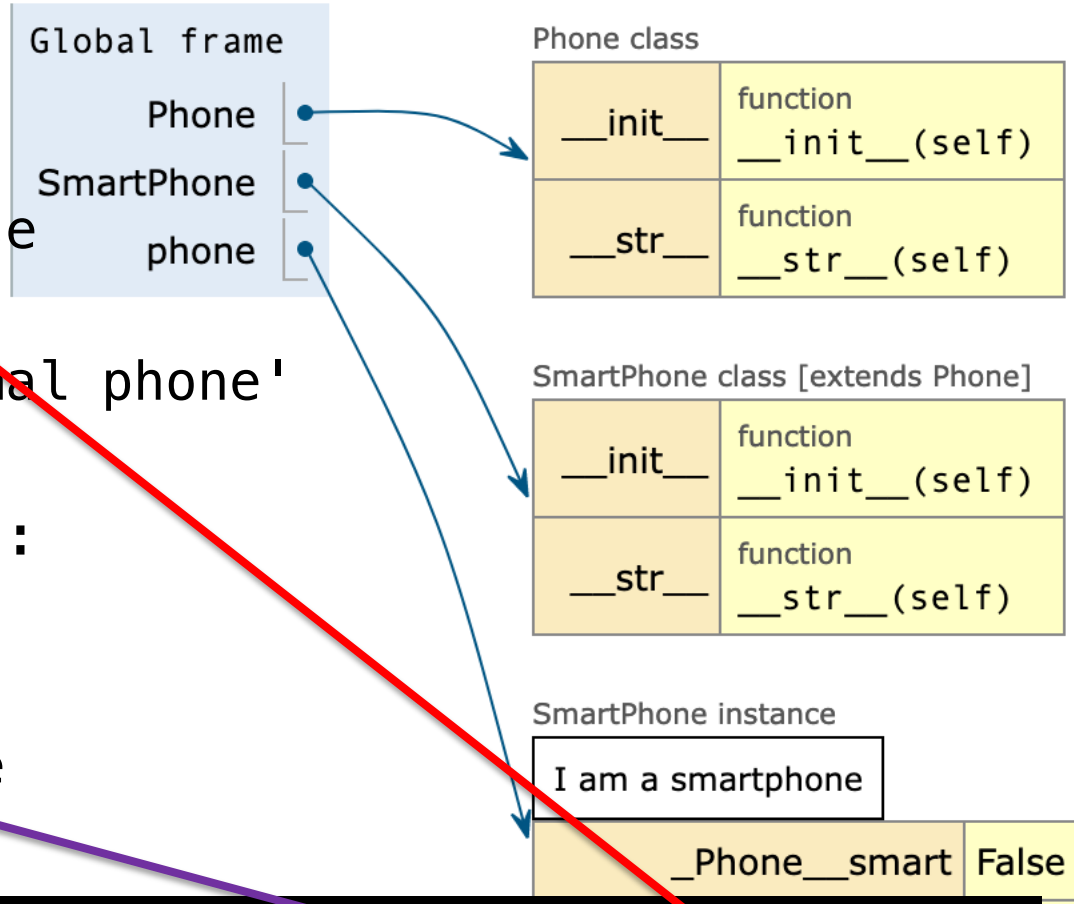



```
>>> phone = SmartPhone()
>>> print(phone)
I am a smartphone
```

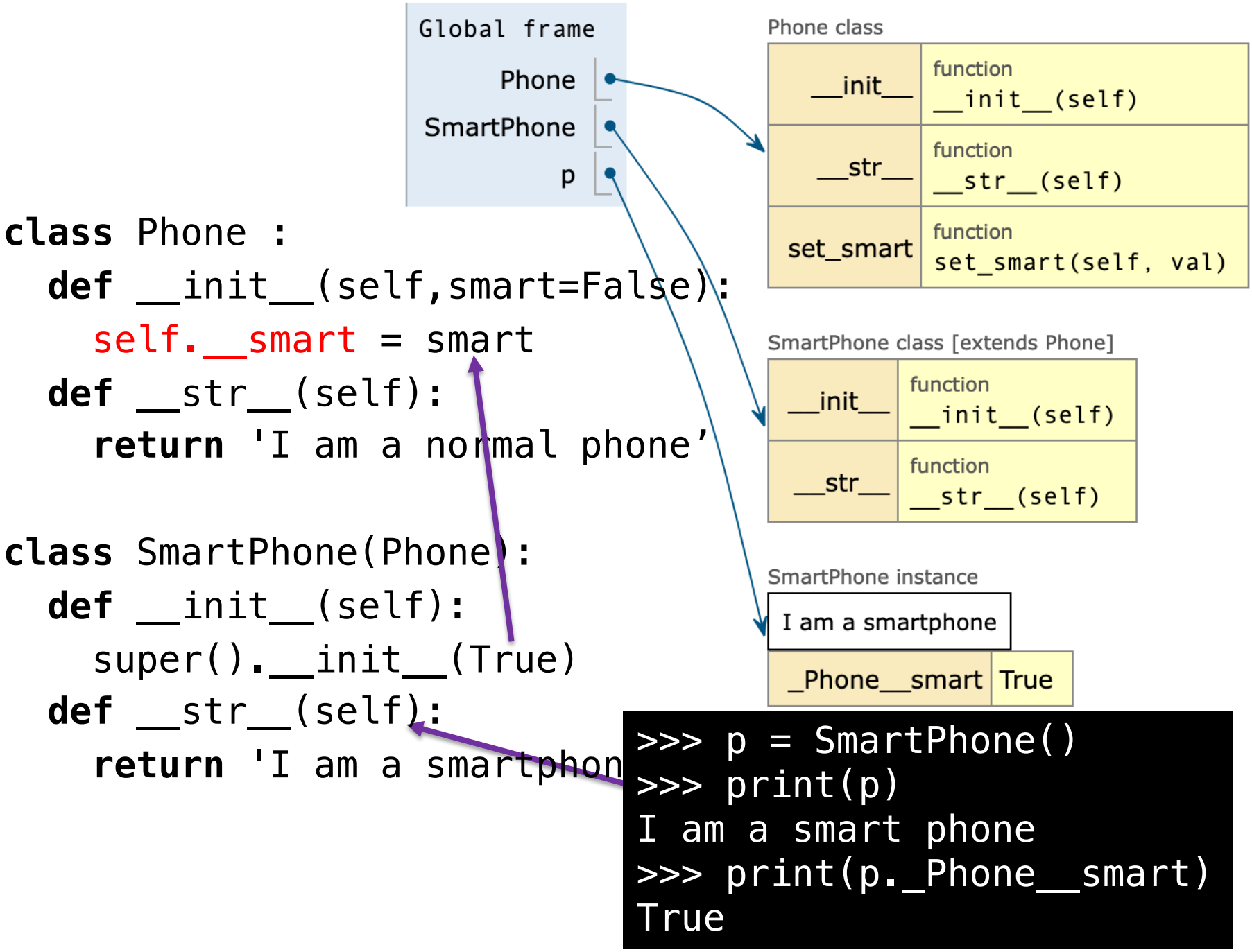
La portée d'une variable d'instance privée

```
class Phone :
  def __init__(self):
    self.__smart = False
  def __str__(self):
    return 'I am a normal phone'

class SmartPhone(Phone):
  def __init__(self):
    super().__init__()
    self.__smart = True
  def __str__(self):
    return 'I am a
```



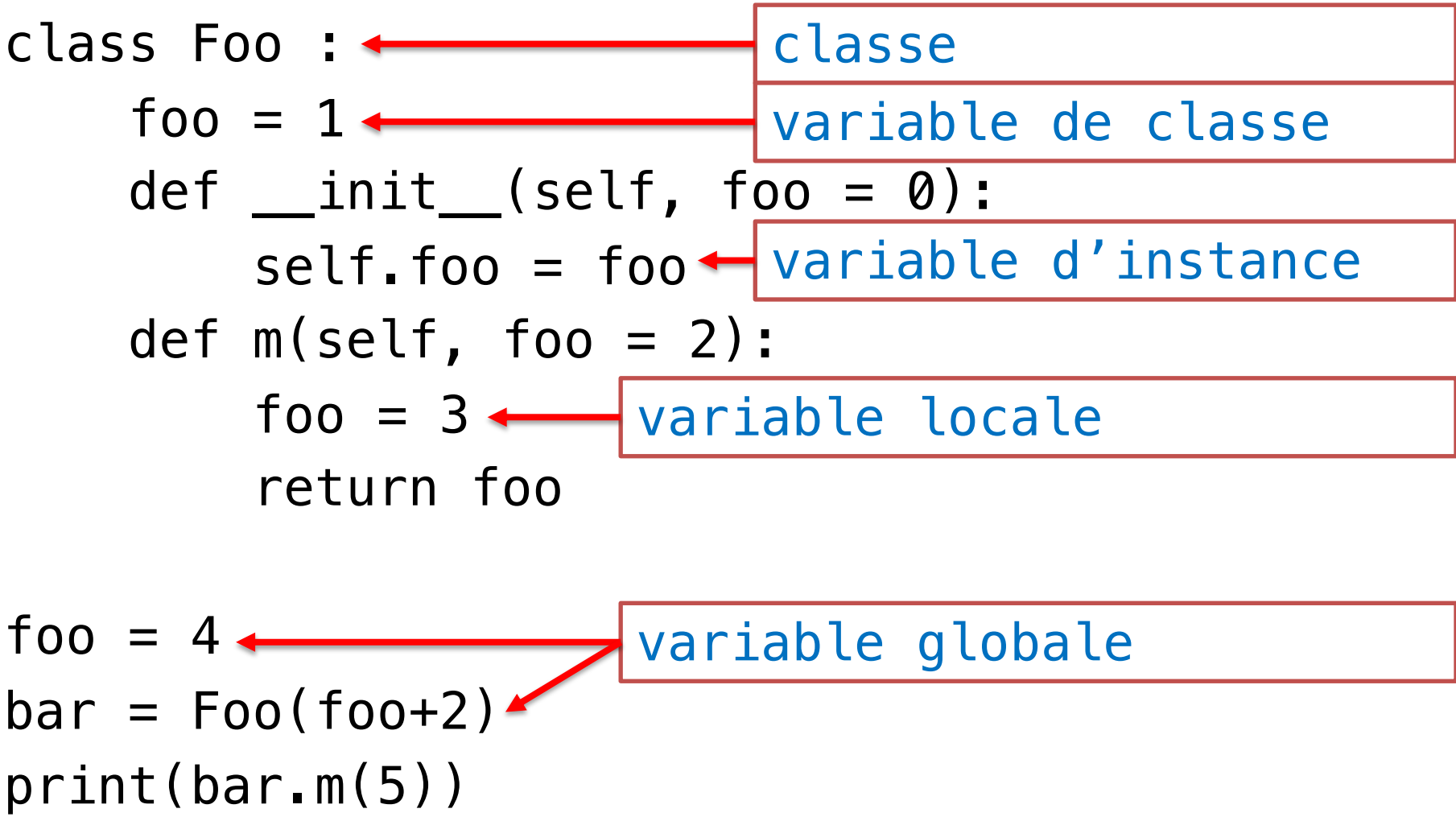

```
>>> p = SmartPhone()
print(p._Phone__smart)      # False
print(p._SmartPhone__smart) # True
```


Global frame
Phone
SmartPhone
p
Phone class
__init__
function
__init__(self)
__str__
function
__str__(self)
set_smart
function
set_smart(self, val)
SmartPhone class [extends Phone]
__init__
function
__init__(self)
__str__
function
__str__(self)
SmartPhone instance
I am a smartphone
_Phone__smart
True
class Phone :
def __init__(self,smart=False):
self.__smart = smart
def __str__(self):
return 'I am a normal phone'
class SmartPhone(Phone):
def __init__(self):
super().__init__(True)
def __str__(self):
return 'I am a smartphon
>>> p = SmartPhone()
>>> print(p)
I am a smart phone
>>> print(p._Phone__smart)
True

La portée des variables

- classe
- variable d’instance
- variable de classe
- variable locale
- variable globale
- paramètres et arguments

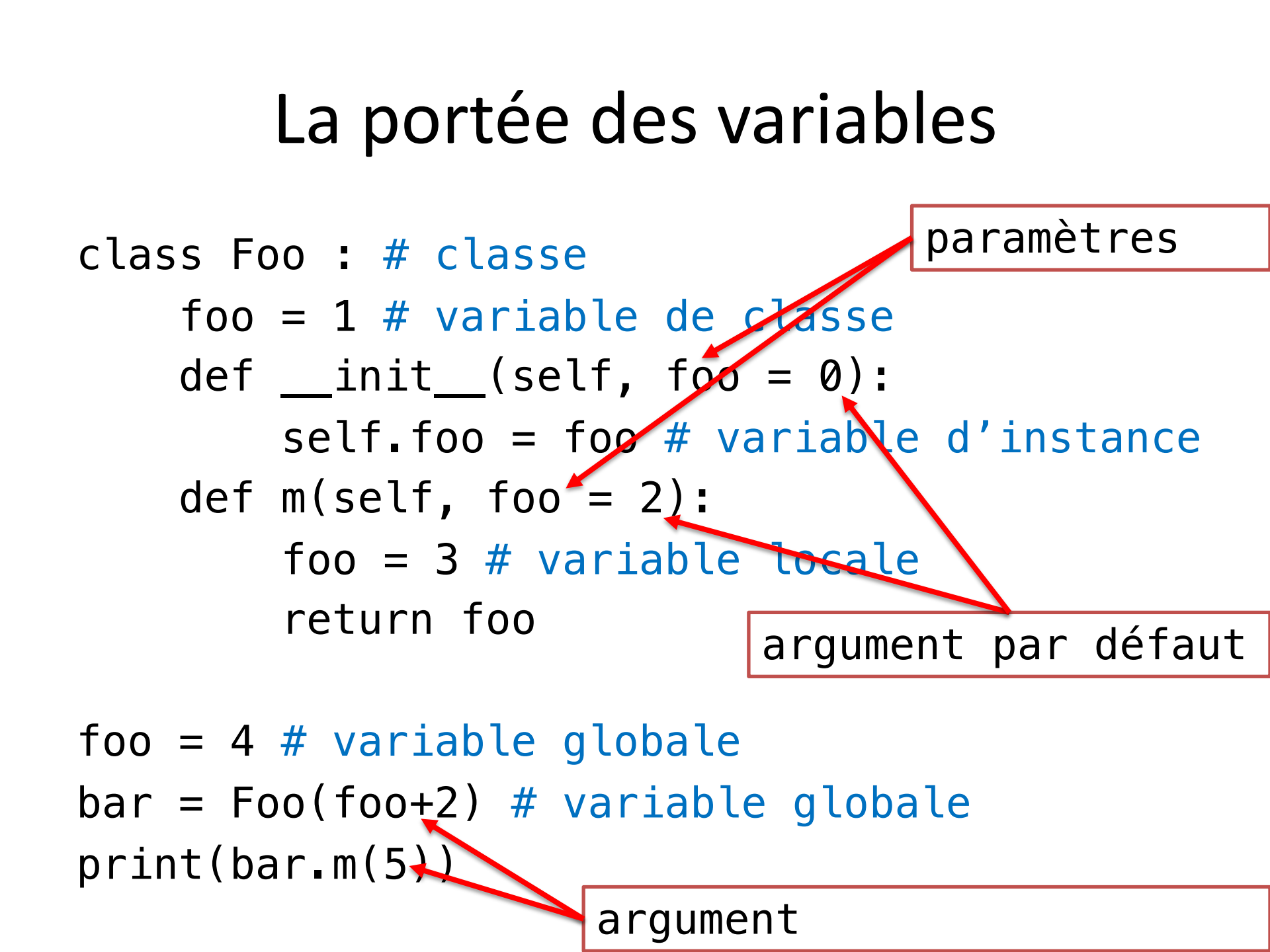
La portée des variables



La portée des variables

```
class Foo : # classe
    foo = 1 # variable de classe
    def __init__(self, foo = 0):
        self.foo = foo # variable d'instance
    def m(self, foo = 2):
        foo = 3 # variable locale
        return foo

foo = 4 # variable globale
bar = Foo(foo+2) # variable globale
print(bar.m(5))
```

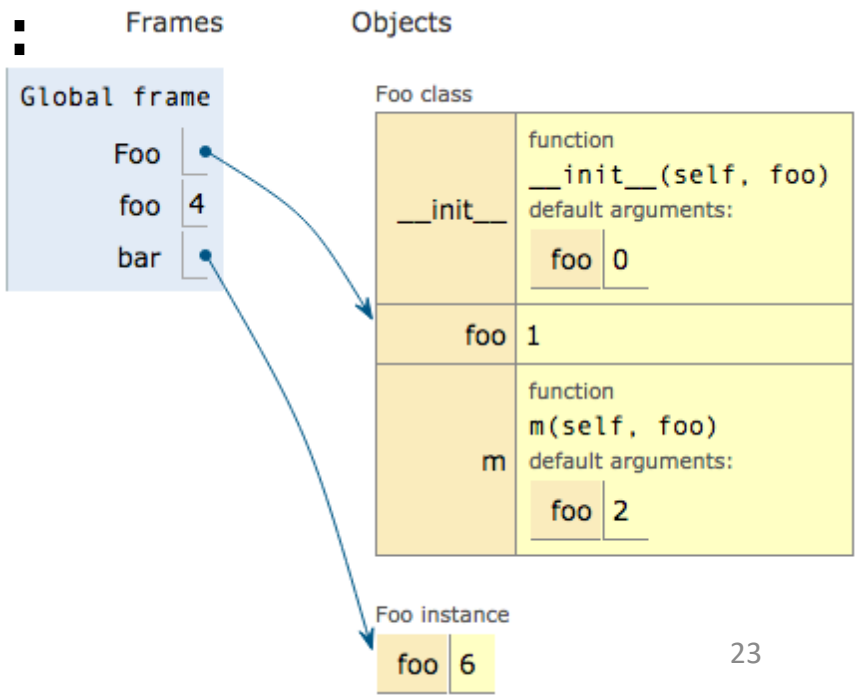


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self, foo = 2):
        foo = 3
        return foo

foo = 4
bar = Foo(foo+2)
print(bar.m(5))
```

?

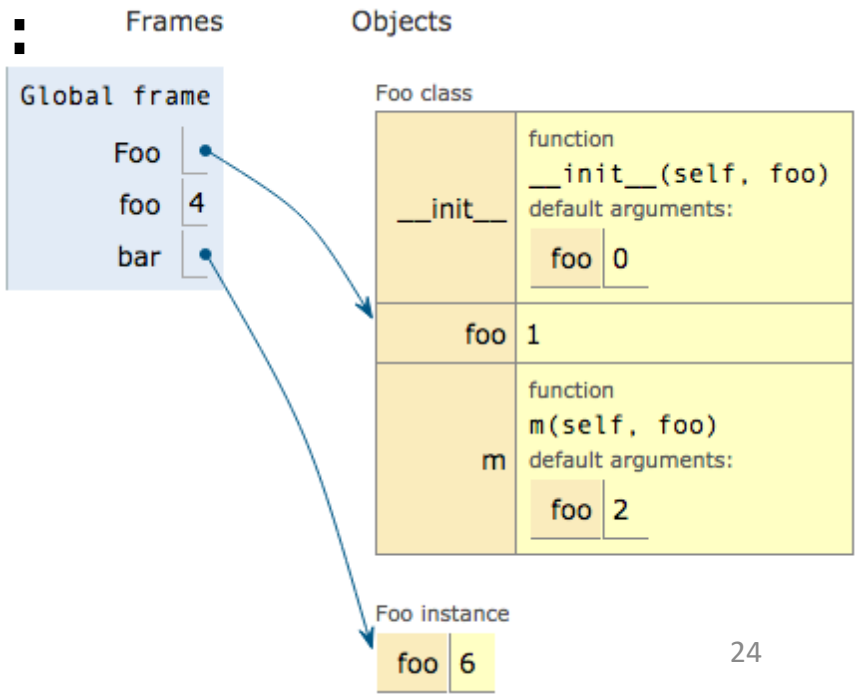


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self, foo = 2):
        foo = 3
        return foo

foo = 4
bar = Foo(foo+2)
print(bar.m(5))
```

3

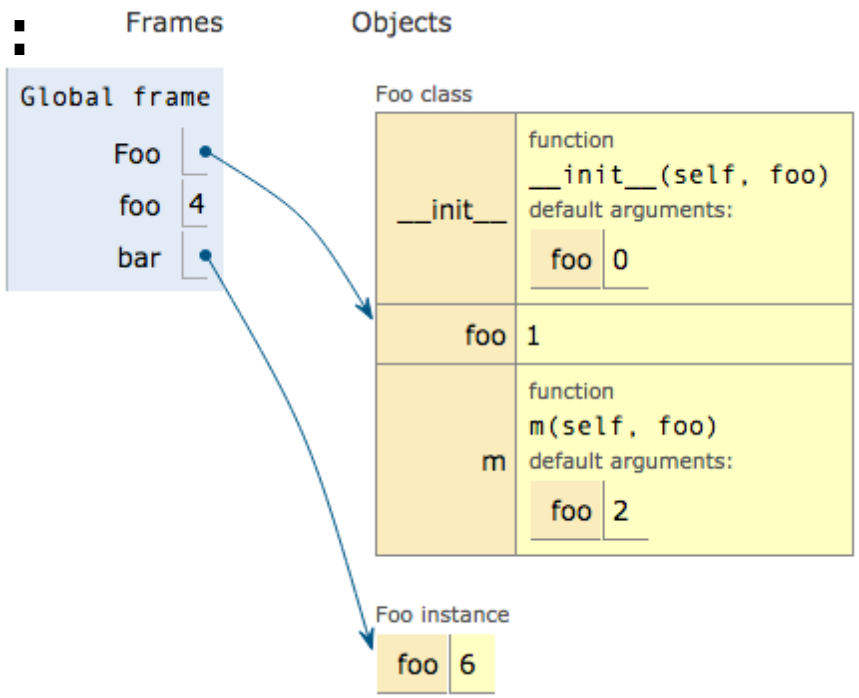


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self, foo = 2):
        #foo = 3
        return foo

foo = 4
bar = Foo(foo+2)
print(bar.m(5))
```

?

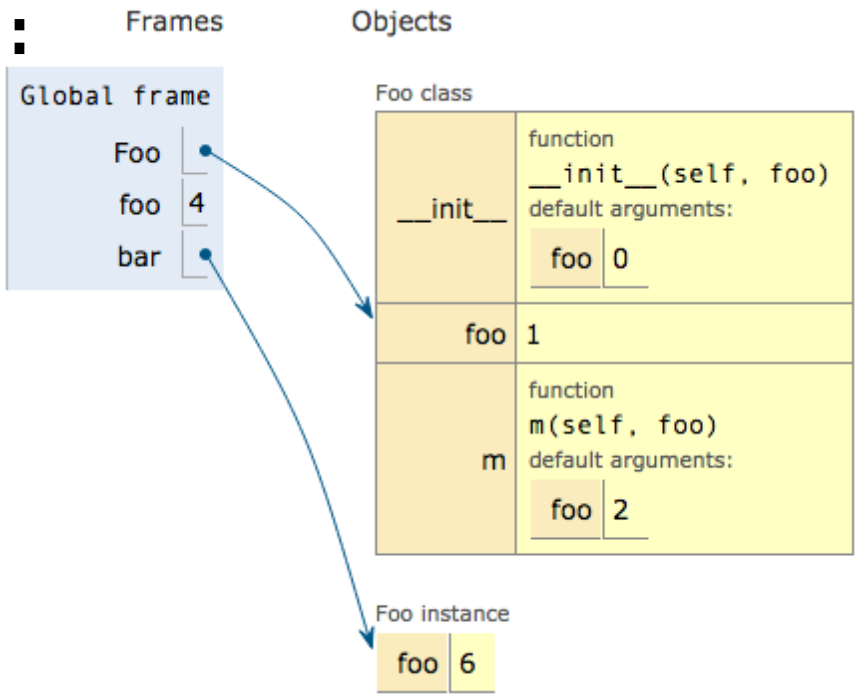


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self, foo = 2):
        return foo

foo = 4
bar = Foo(foo+2)
print(bar.m(5))
```

5

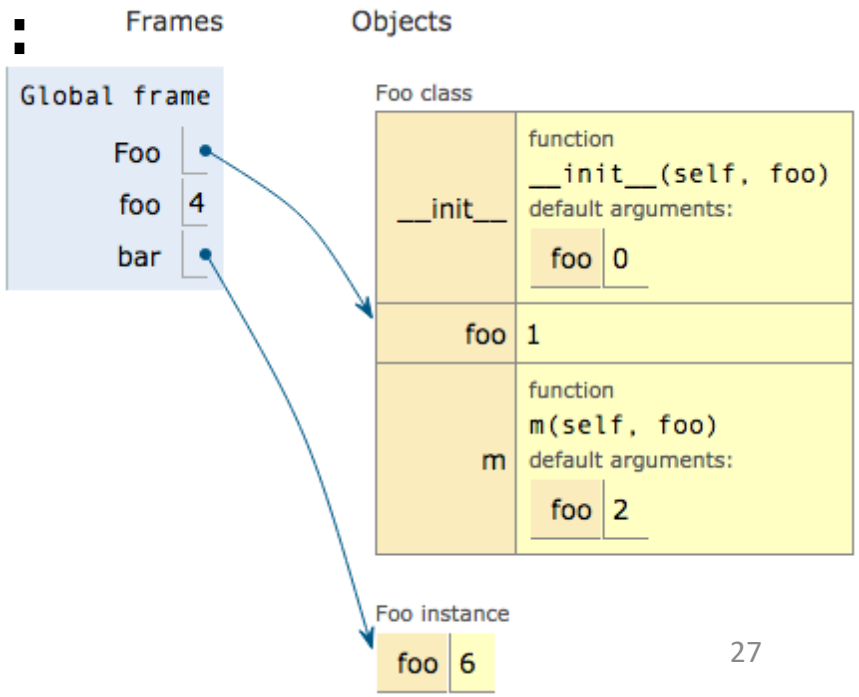


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self, foo = 2):
        return foo

foo = 4
bar = Foo(foo+2)
print(bar.m())
```

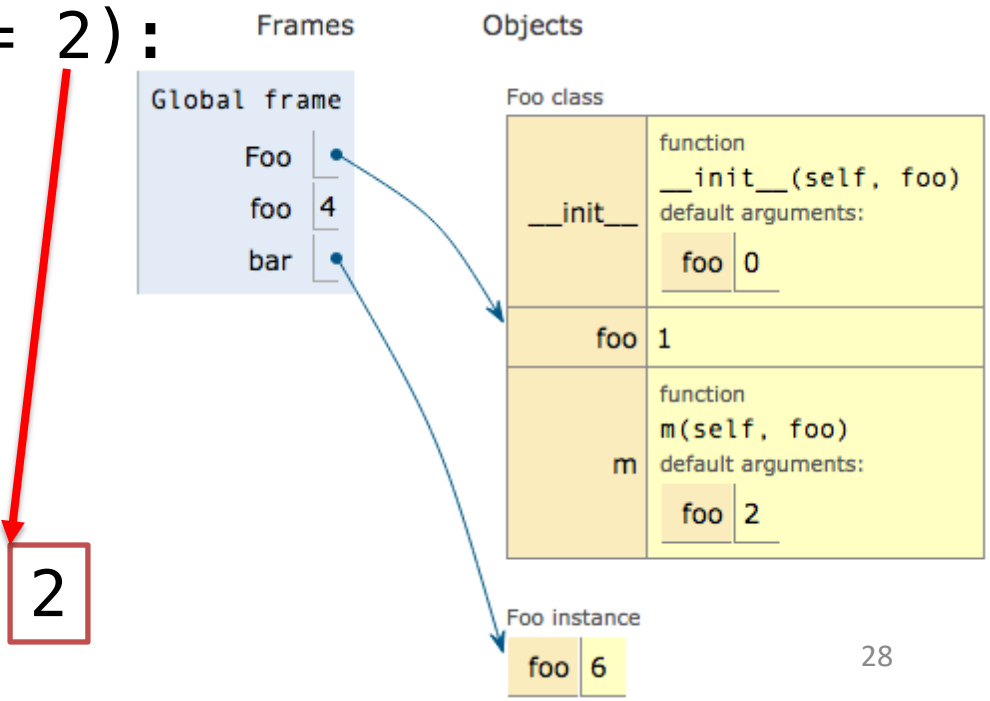
?



La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self, foo = 2):
        return foo

foo = 4
bar = Foo(foo+2)
print(bar.m())
```

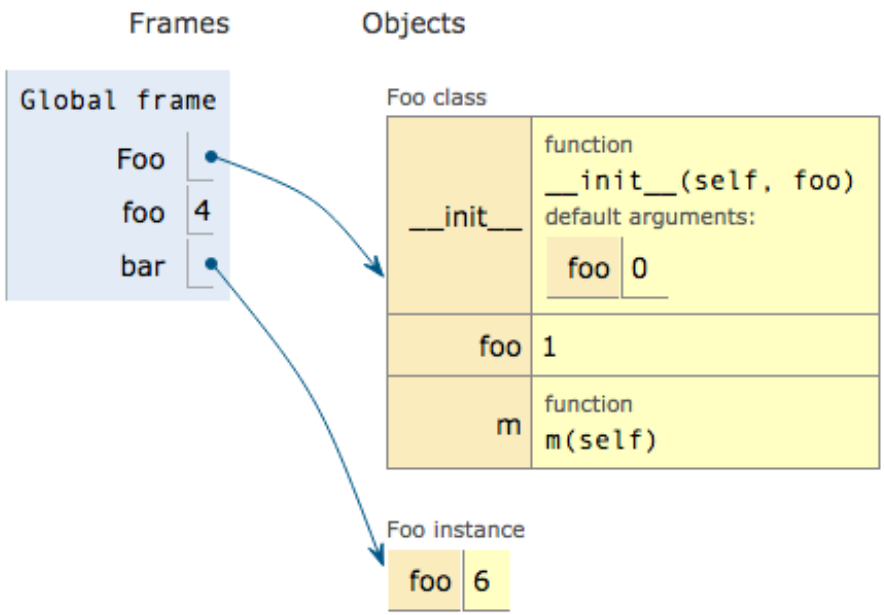


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self, foo = 2):
        return foo

foo = 4
bar = Foo(foo+2)
print(bar.m())
```

?

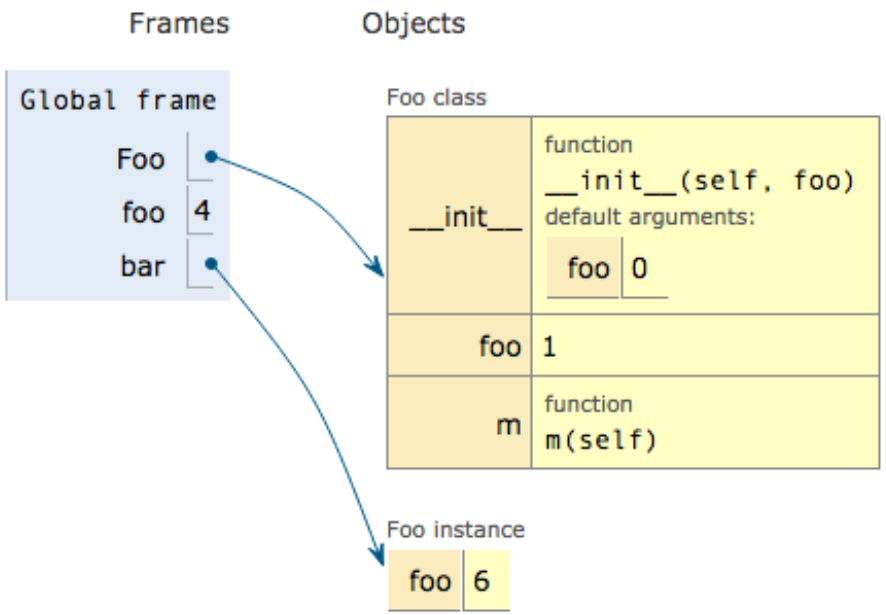


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self):
        return foo

foo = 4
bar = Foo(foo+2)
print(bar.m())
```

4

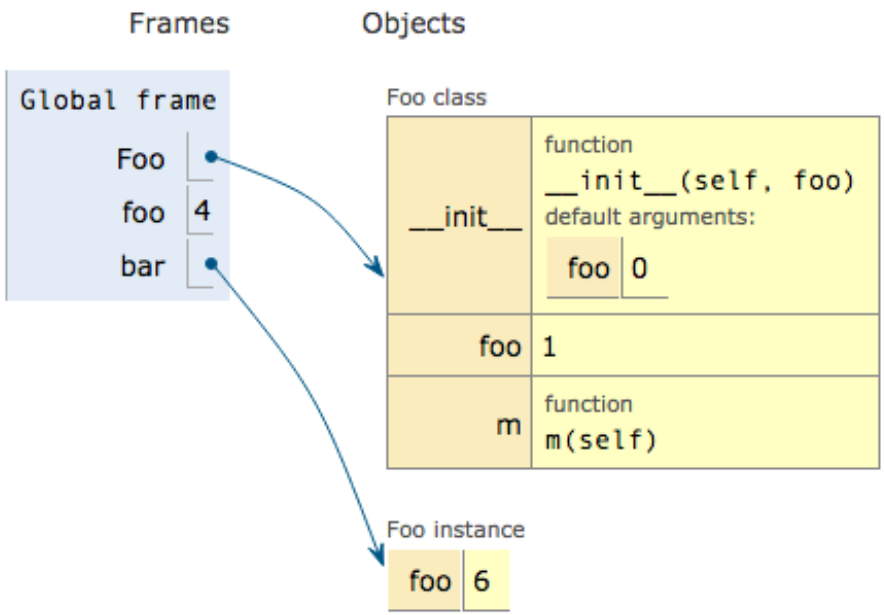


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self):
        return self.foo

foo = 4
bar = Foo(foo+2)
print(bar.m())
```

?

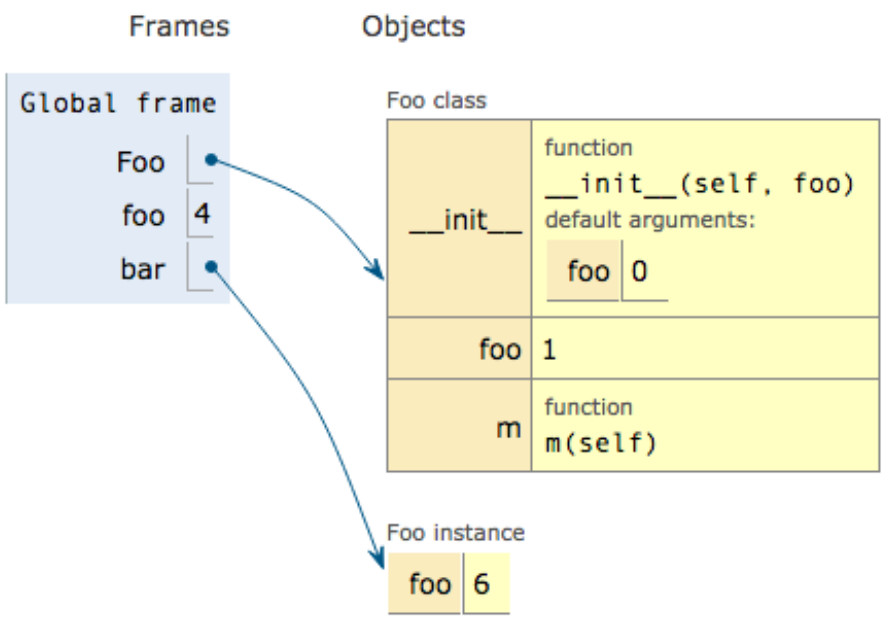


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self):
        return self.foo

foo = 4
bar = Foo(foo+2)
print(bar.m())
```

6

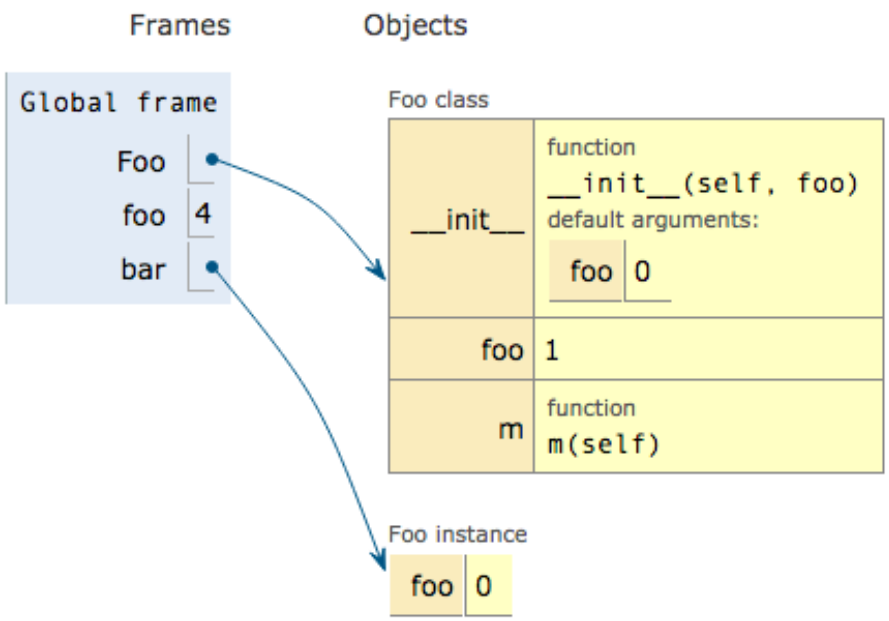


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self):
        return self.foo

foo = 4
bar = Foo()
print(bar.m())
```

?

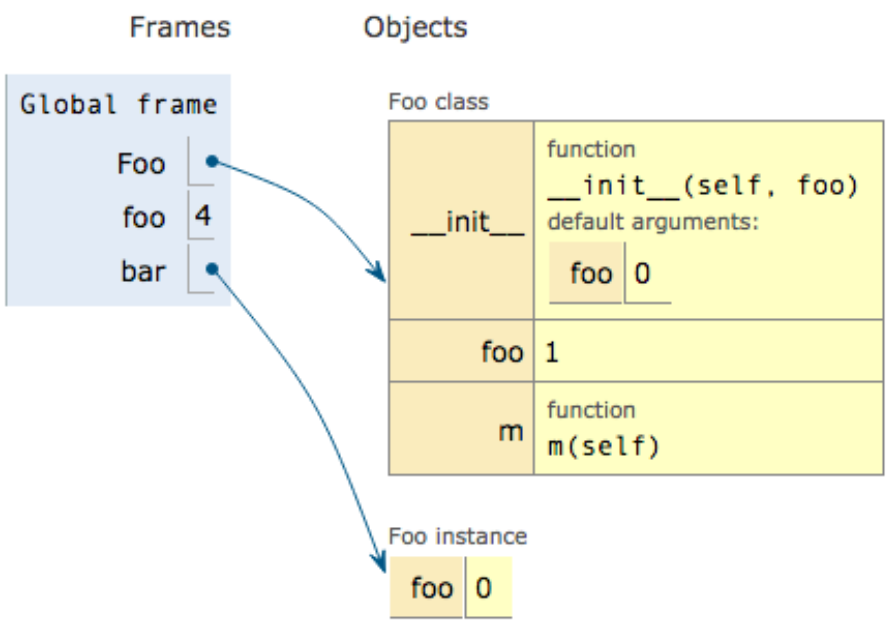


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self):
        return self.foo

foo = 4
bar = Foo()
print(bar.m())
```

0

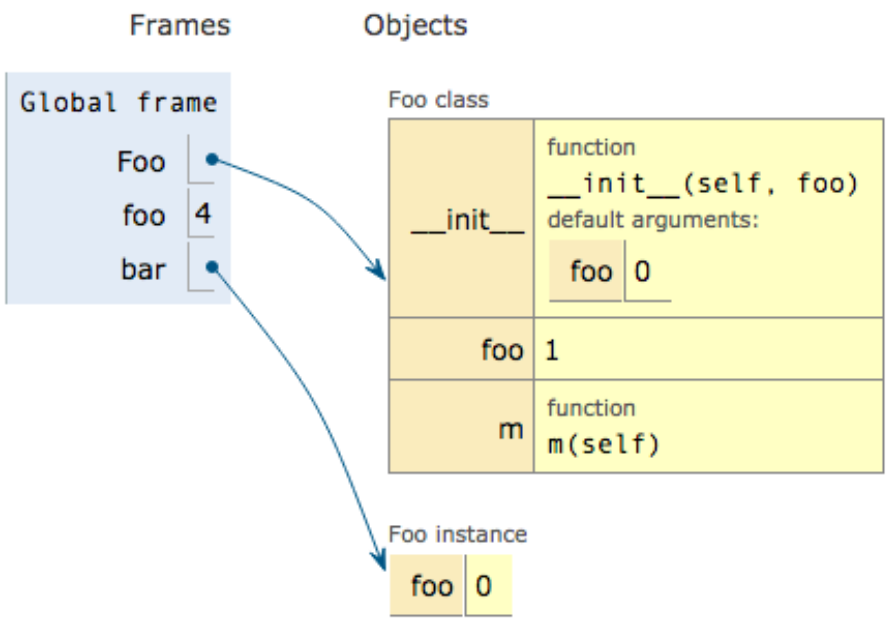


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self):
        return Foo.foo

foo = 4
bar = Foo()
print(bar.m())
```

?

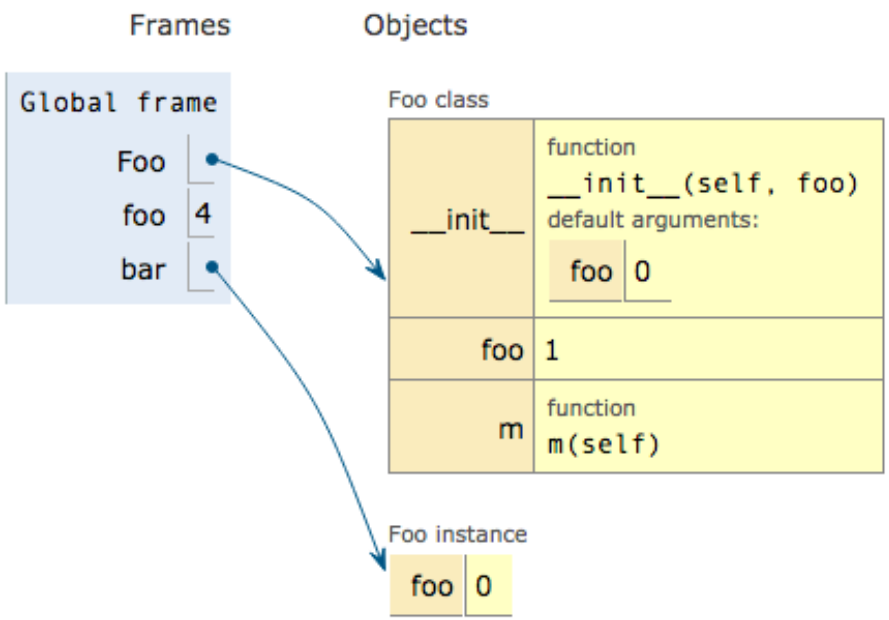


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        self.foo = foo
    def m(self):
        return Foo.foo

foo = 4
bar = Foo()
print(bar.m())
```

1

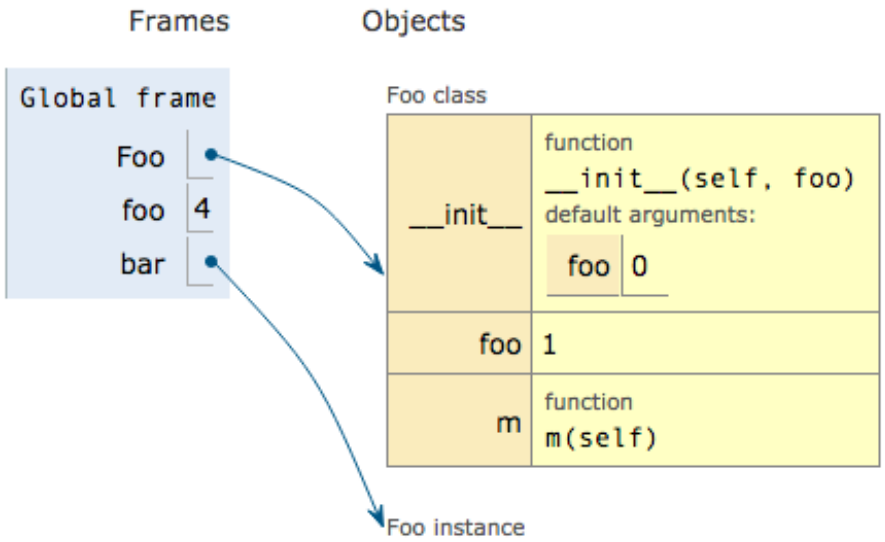


La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        #self.foo = foo
        pass
    def m(self):
        return self.foo

foo = 4
bar = Foo()
print(bar.m())
```

?



La portée des variables

```
class Foo :
    foo = 1
    def __init__(self, foo = 0):
        pass
    def m(self) :
        return self.foo

foo = 4
bar = Foo()
print(bar.m())
```

1