Python Programming Language Foundation

Session 6

Session overview

Object Oriented Programming

- Inheritance in Python
- Polymorphism in Python
- Encapsulation in Python

Class-related decorators

- @classmethod
- @staticmethod
- @abstractmethod
- @property

Programming paradigms Python supports

Procedural

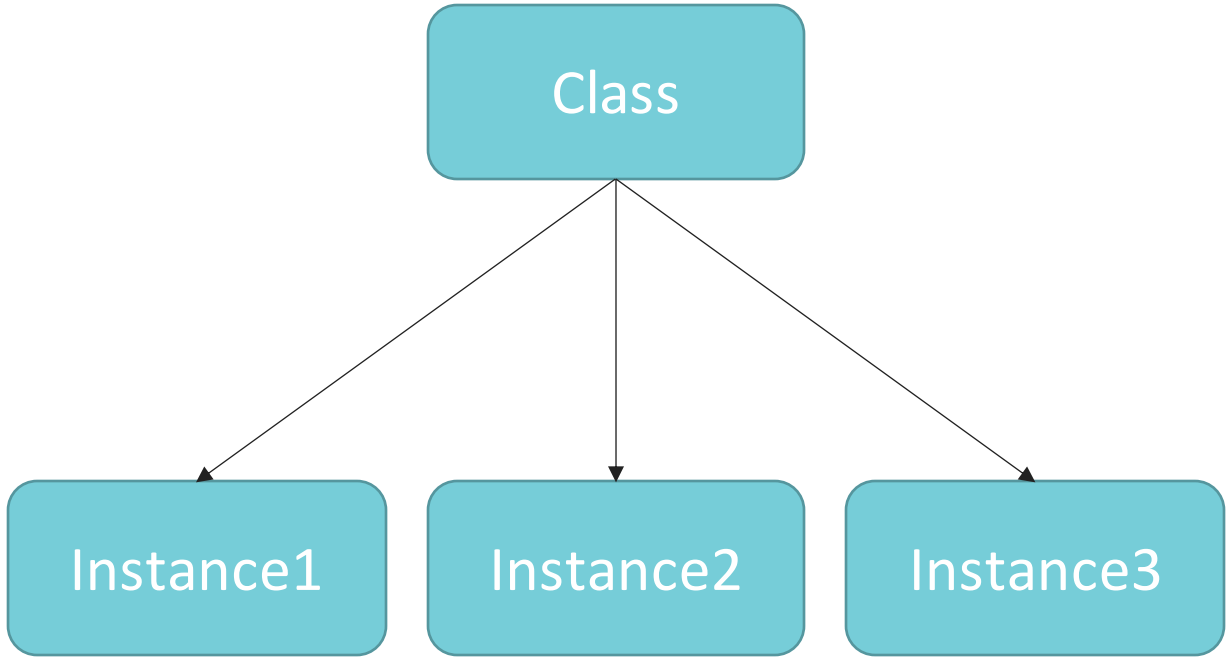
Functional

Object-Oriented

Object Oriented Programming

OOP, which stands for Object-oriented programming, is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual *objects*.

Difference class and instance



Class definition

```
class Monkey:
    """Just a little monkey."""
    banana_count = 5

    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f'Hi, I am {self.name}!')

    def eat_banana(self):
        if self.banana_count > 0:
            self.banana_count -= 1
            print('Yammy!')
        else:
            print('Still hungry :(')
```

```
>>> travor_monkey = Monkey('Travor')
>>> daniel_monkey = Monkey('Daniel')
>>> travor_monkey.greet()
'Hi, I am Travor!'
```

```
>>> travor_monkey is daniel_monkey
False
>>> travor_monkey is Monkey
False
>>> travor_monkey is Monkey('Travor')
False
```

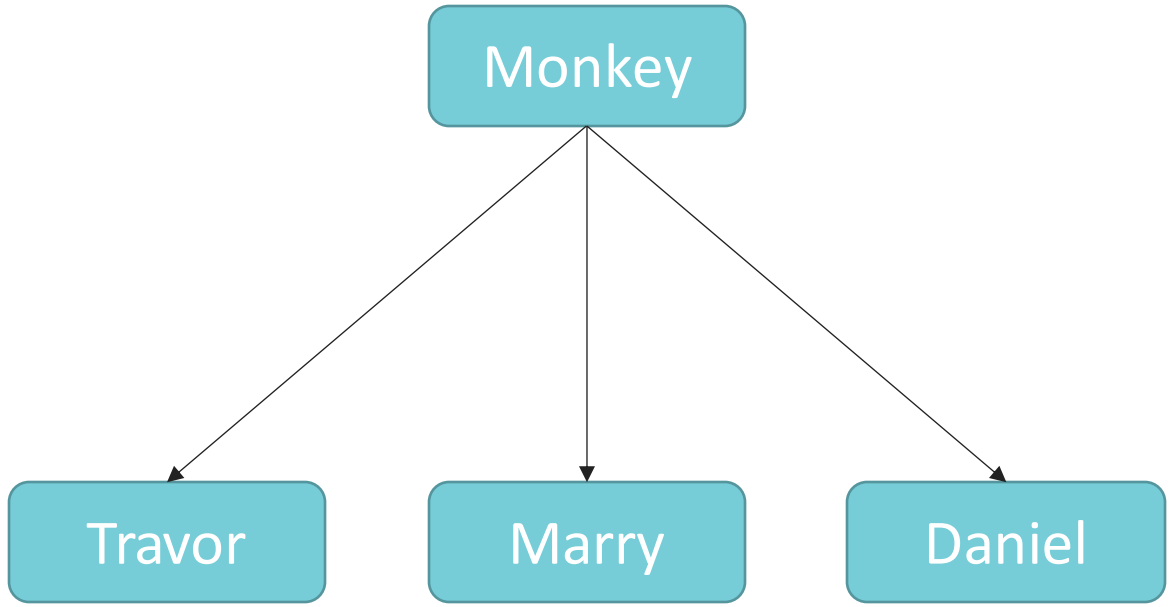
Class definition

```
class Monkey:
    """Just a little monkey."""
    banana_count = 5

    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f'Hi, I am {self.name}!')

    def eat_banana(self):
        if self.banana_count > 0:
            self.banana_count -= 1
            print('Yammy!')
        else:
            print('Still hungry :(')
```

```
>>> travor_monkey.eat_banana()
'Yammy'


>>> print(travor_monkey.banana_count)
4
>>> print(Monkey.banana_count)
5
>>> print(daniel_monkey.banana_count)
5
```

Difference between class object and instance object

Class object



Instance objects

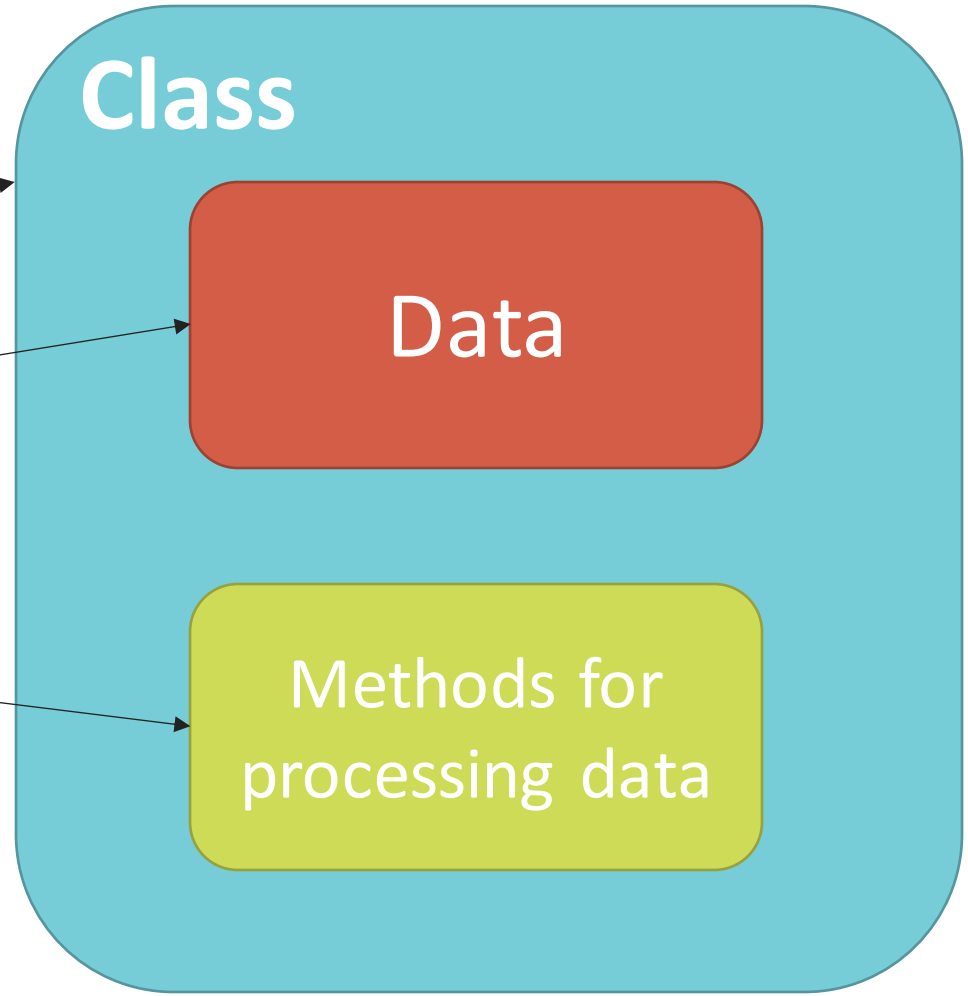
Object-Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

Encapsulation

```
class Five:

    value = 5

    def print_value(self):
        print(self.value)
```



Data hiding

```
class Person:

    def __init__(self, name, age, salary, friends):
        self.name = name
        self._age = age
        self.__salary = salary
        self.__friends__ = friends

    def print_info(self):
        print(self.name)
        print(self._age)
        print(self.__salary)
        print(self.__friends__)
```

Data hiding

```
>>> alice = Person(
        'Alice Doe',
        age=42,
        salary=500,
        friends=None,
    )


>>> alice.print_info()
'Alice Doe'
42
500
None
```

```
>>> print(alice.name)
'Alice Doe'

>>> print(alice._age)
42

>>> print(alice.__salary)
AttributeError: 'Person' object has
no attribute '__salary'

>>> print(alice.__friends__)
None

>>> print(alice._Person__salary)
500
```

Inheritance

Inheritance usage

```
class Ancestor:
    def __init__(self):
        print("Ancestor.__init__")

    def fun(self):
        print("Ancestor.fun")

    def work(self):
        print("Ancestor.work")
```

```
class Child(Ancestor):
    def __init__(self):
        print("Child.__init__")

    def fun(self):
        print("Child.fun")
```

Inheritance usage

```
>>> from tmp import Child

>>> c = Child()
Child.__init__

>>> c.fun()
Child.fun

>>> c.work()
Ansestor.work
```

super([type, [object]])

Return a proxy object that delegates method calls to a parent or sibling class of type. This is useful for accessing inherited methods that have been overridden in a class.

Documentation: https://docs.python.org/3.6/library/functions.html#super

Inheritance and `super()` built-in

```
class Ancestor:
    def __init__(self):
        print("Ancestor.__init__")

    def fun(self):
        print("Ancestor.fun")
```

```
class Child(Ancestor):
    def __init__(self):
        super().__init__()
        print("Child.__init__")

    def fun(self):
        super().fun()
        print("Child.fun")
```

Inheritance and `super()` built-in

```
>>> from tmp import Child

>>> c = Child()
Ancestor.__init__
Child.__init__

>>> c.fun()
Ancestor.fun
Child.fun
```

Diamond problem

Diamond problem

```
class Ancestor:
    def __init__(self):
        print("Ancestor.__init__")

    def fun(self):
        print("Ancestor.fun")
```

```
class Child1(Ancestor):
    def __init__(self):
        print("Child1.__init__")
        super().__init__()
```

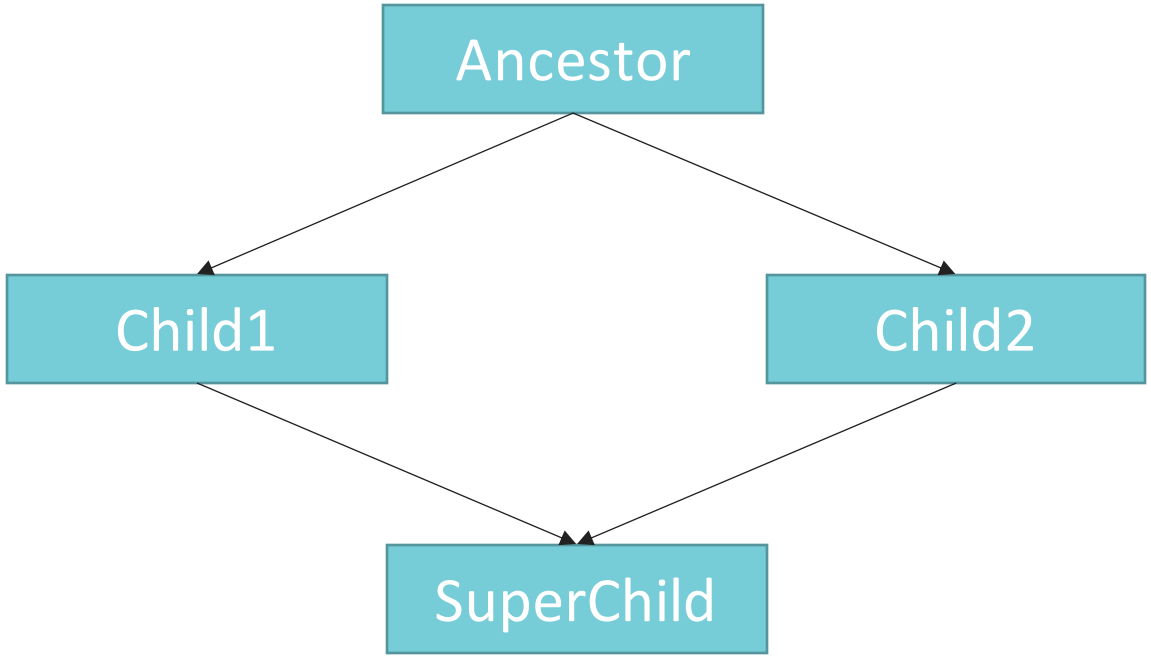
```
class Child2(Ancestor):
    def __init__(self):
        print("Child2.__init__")
        super().__init__()
```

Diamond problem

```
class SuperChild(Child1, Child2):
    def __init__(self):
        print("SuperChild.__init__")
        super().__init__()
```

Diamond problem

```
>>> c = SuperChild()

SuperChild.__init__
Child1.__init__
Child2.__init__
Ancestor.__init__
```

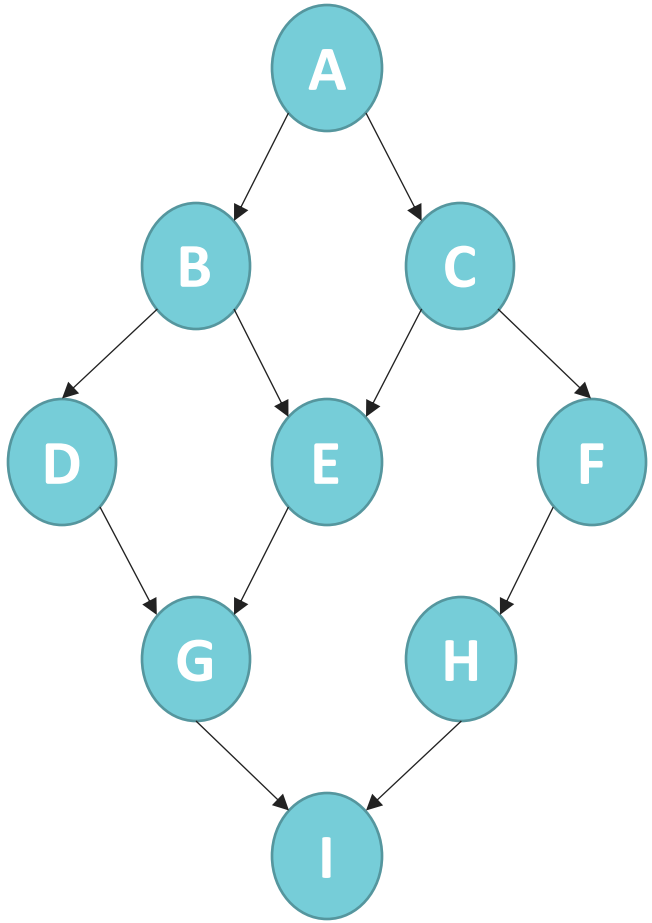


Diamond problem

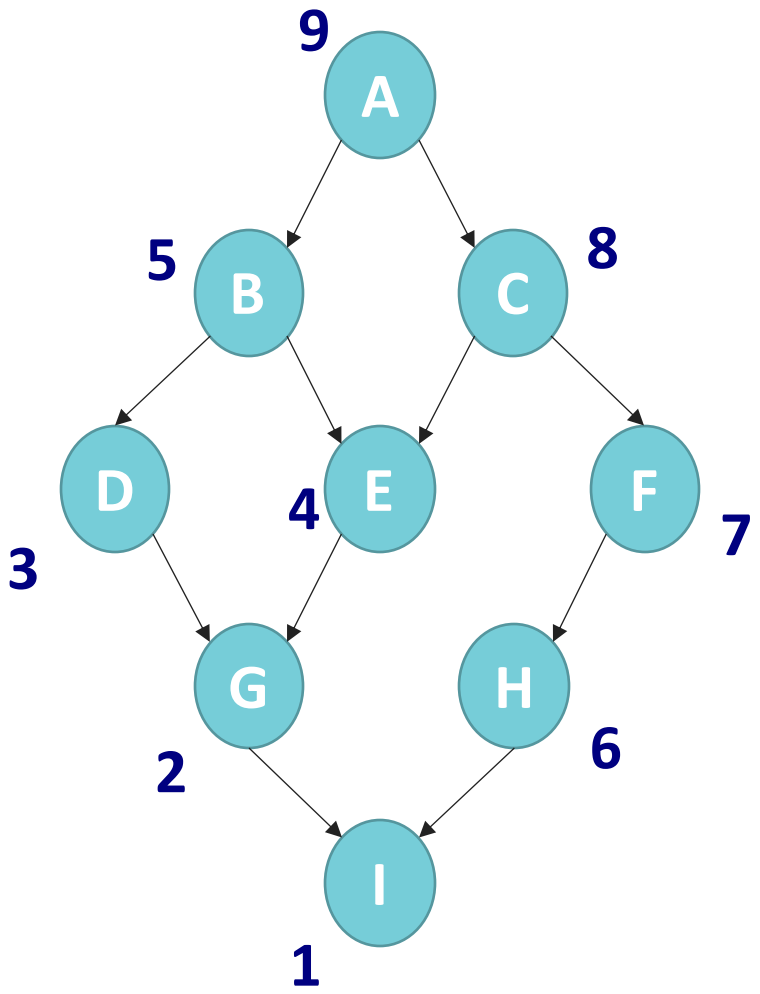
Method Resolution Order (MRO) is the order in which Python looks for a method in a hierarchy of classes. Especially it plays vital role in the context of multiple inheritance as single method may be found in multiple super classes.

So what is the problem here?…

Diamond problem

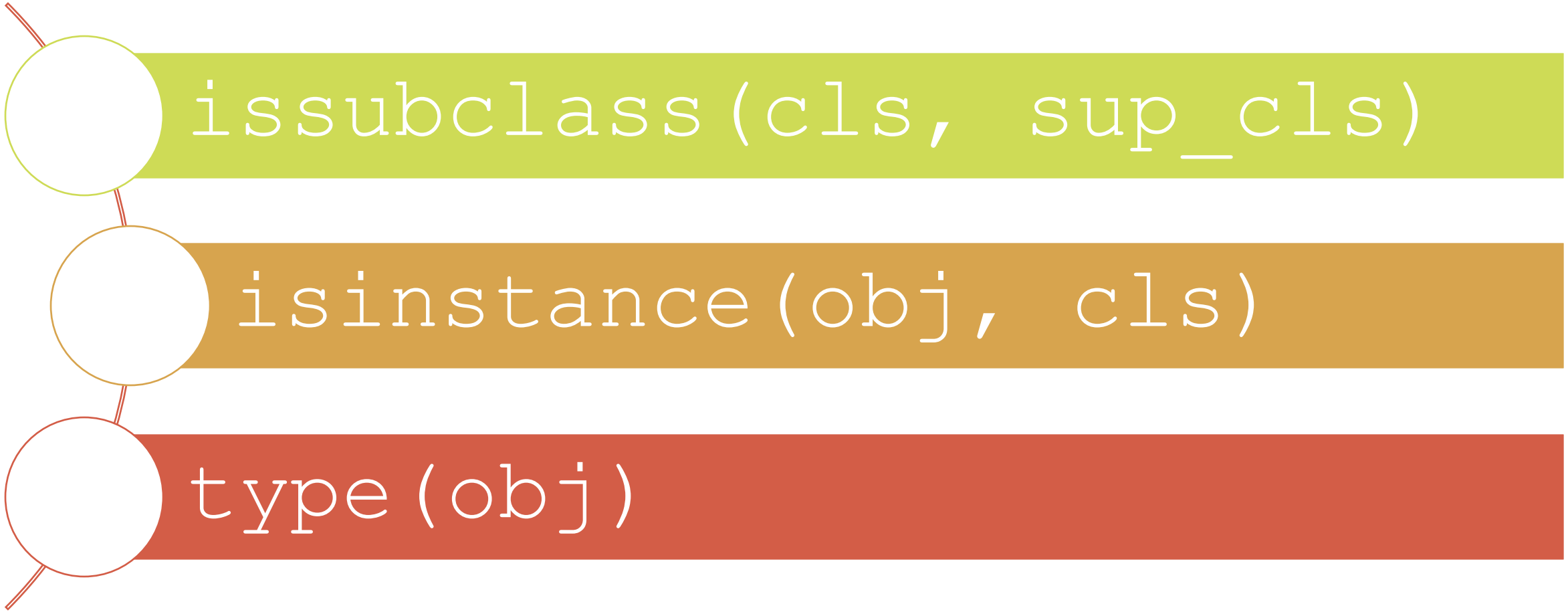


Diamond problem




```
__mro__:
I,G,D,E,B,H,F,C,A,Object
```

Relationships between classes



'isinstance' vs 'type'

```
class A:
    pass


a = A()
o = object()
```

```
>>> print(isinstance(a, A))
True

>>> print(isinstance(a, object))
True

>>> print(isinstance(o, A))
False

>>> print(type(a) is A)
True

>>> print(type(a) is object)
False
```

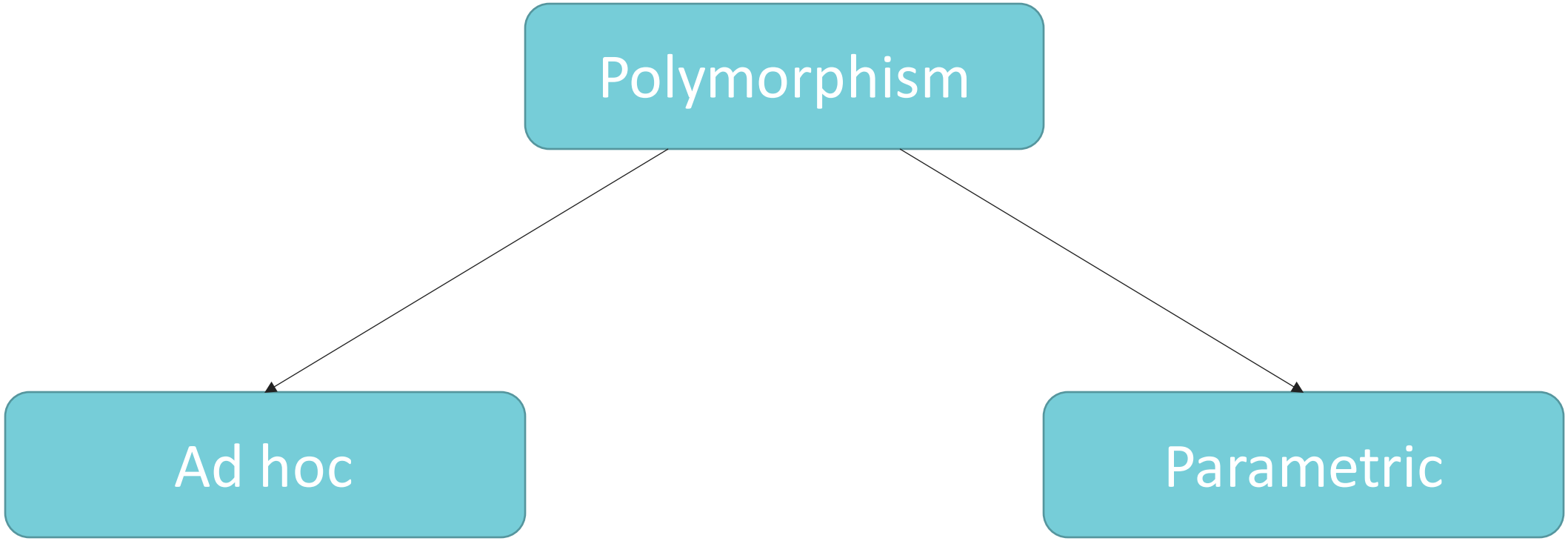
'issubclass' built-in

```
class A:
    pass


class B(A):
    pass


class C:
    pass
```

```
>>> print(issubclass(B, A))
True

>>> print(issubclass(A, B))
False

>>> print(issubclass(A, C))
False
```

Polymorphism

Polymorphism



Ad hoc polymorphism

C++ language example:

```
class MySum():
{
    public:
    double sum(double a, double b)
    {
        return a + b;
    }

    double sum(int a, int b, int c)
    {
        return double(a + b + c);
    }
}
```

Python language example:

```
class MySum:

    def sum(self, a, b)
        return a + b

    def sum(self, a, b, c)
        return a + b + c


>>> ms = MySum()
>>> ms.sum(1,2,3)
6
>>> ms.sum(1,2)
TypeError: sum() missing 1
required positional argument: 'c'
```

Python example:

```
>>> 1 + 1
2
>>> 1 + True
2
>>> 1 + 1.0
2.0
```

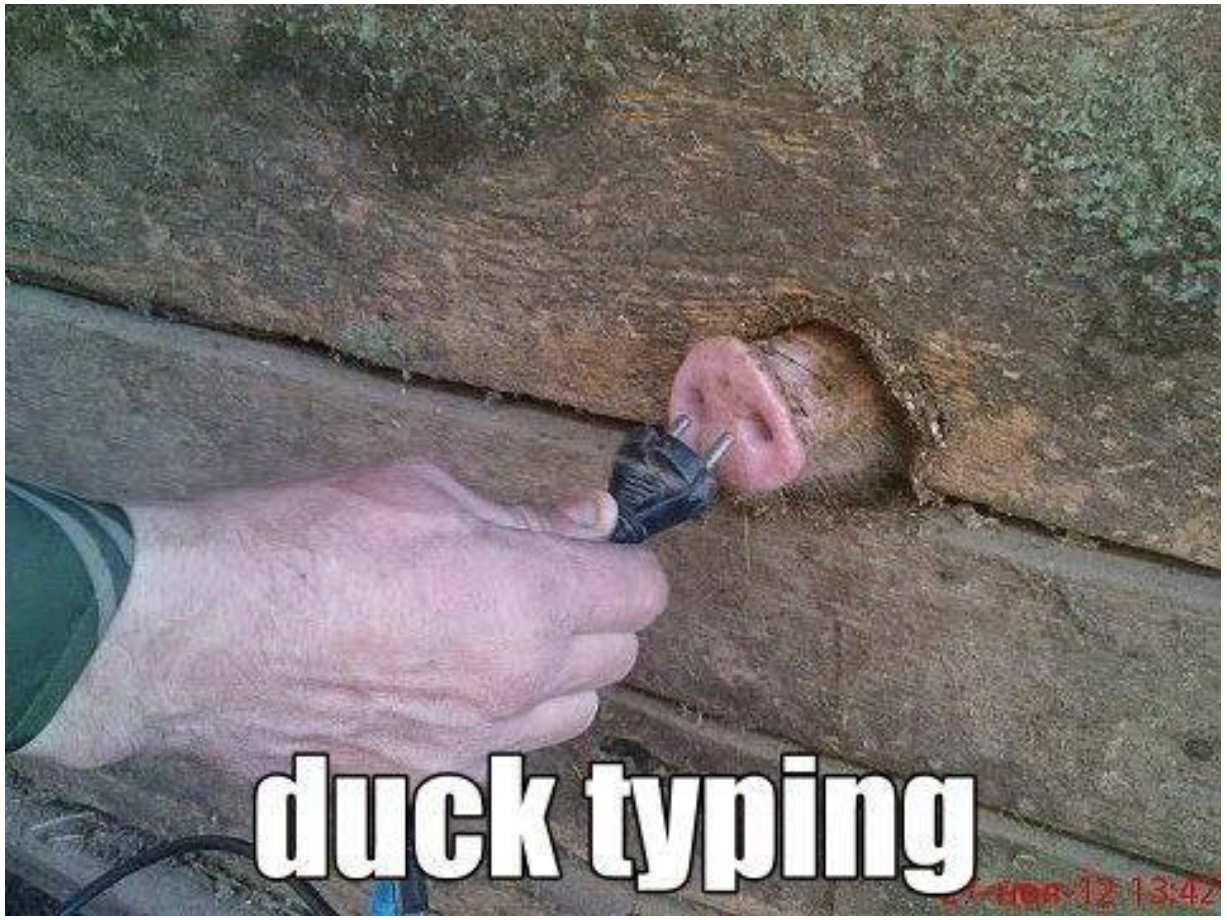
Parametric polymorphism

```
1 + 1       <⟹>  int(1).__add__(1)
1 + True    <⟹>  int(1).__add__(True)
1 + 1.0     <⟹>  int(1).__add__(1.0)
```

Duck typing	application of the duck test to determine if an object can be used for a particular purpose

"If it walks like a duck and it quacks like a duck then it must be a duck"

Duck typing



Duck typing

```
class Duck:
    def fly(self):
        print("Duck flying")


class Airplane:
    def fly(self):
        print("Airplane flying")


class Whale:
    def swim(self):
        print("Whale swimming")
```

```
def lift_off(entity):
    entity.fly()

duck = Duck()
airplane = Airplane()
whale = Whale()

>>> lift_off(duck)
Duck flying

>>> lift_off(airplane)
Airplane flying

lift_off(whale)
AttributeError: 'Whale' object
has no attribute 'fly'
```

Operators override

```
class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self, other):
        return Vector(self.a + other.a, self.b + other.b)


>>> v1 = Vector(2, 10)
>>> v2 = Vector(5, -2)
>>> print(v1 + v2)
'Vector (7, 8)'
```

Magic Methods

Magic methods

```
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length',
'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real',
'to_bytes']
```

Magic methods

Syntax	Method	Operation
`a + b`	`a.__add__(b)`	Addition
`a - b`	`a.__sub__(b)`	Subtraction
`a * b`	`a.__mul__(b)`	Multiplication
`a / b`	`a.__truediv__(b)`	Division
`a % b`	`a.__mod__(b)`	Modulus
`a ** b`	`a.__pow__(b)`	Exponent
`a // b`	`a.__floordiv__(b)`	Floor division

Magic methods

Syntax	Method	Operation
`a == b`	`a.__eq__(b)`	Equality
`a != b`	`a.__ne__(b)`	Difference
`a < b`	`a.__lt__(b)`	Ordering
`a <= b`	`a.__le__(b)`	
`a >= b`	`a.__gt__(b)`	
`a > b`	`a.__gt__(b)`	

__add__

```
class CustomNumber:
    def __init__(self, number):
        self._number = number
    def __add__(self, number):
        print("I'm counting!")
        return CustomNumber(self._number + number)


>>> number = CustomNumber(10)

>>> number + 10
'I'm counting!'
'<__main__.CustomNumber object at 0x111207e80>'
```

```
class Dog:
    def __init__(self, name):
        self.name = name

>>> dog= Dog('Snow')


>>> print(dog)
'<__main__.Dog object at 0x111207e80>'

>>> dog
'<__main__.Dog object at 0x111207e80>'
```

```
class ReadableDog(Dog):
    def __str__(self):
        return f'Dog is named {self.name}'


>>> dog = ReadableDog('Snow')


>>> print(dog)
'Dog is named Snow'

>>> dog
'<__main__.ReadableDog object at 0x111207e80>'
```

__repr__

```
class ReadableDog(Dog):
    def __str__(self):
        return f'Dog is named {self.name}'
    def __repr__(self):
        return f'ReadableDog(name="{self.name}")'

>>> dog = ReadableDog('Snow')

>>> print(dog)
Dog is named Snow'

>>> dog
'ReadableDog(name="Snow")'
```

__repr__

```
class ReadableDog(Dog):
    def __repr__(self):
        return f'ReadableDog(name="{self.name}")'
```

```
>>> dog = ReadableDog('Snow')

>>> print(dog)
'ReadableDog(name="Snow")'

>>> dog
'ReadableDog(name="Snow")'
```

__repr__ vs __str__

- __str__() representation is *user-friendly* string.

- __repr__() representation is for *developers* so they can use it to *debug*.

__getitem__

```
class CustomContainer:

    def __init__(self, internal_list):
        self._internal_list = internal_list

    def __getitem__(self, item):
        internal_item = self._internal_list[item]
        return str(internal_item)


>>> my_container = CustomContainer([1, '2', 3, 4, 5])

>>> container[2]
'3'
```

Standard Class-related Decorators

Class-related decorators

@classmethod

@staticmethod

@abstractmethod

@property

@classmethod decorator

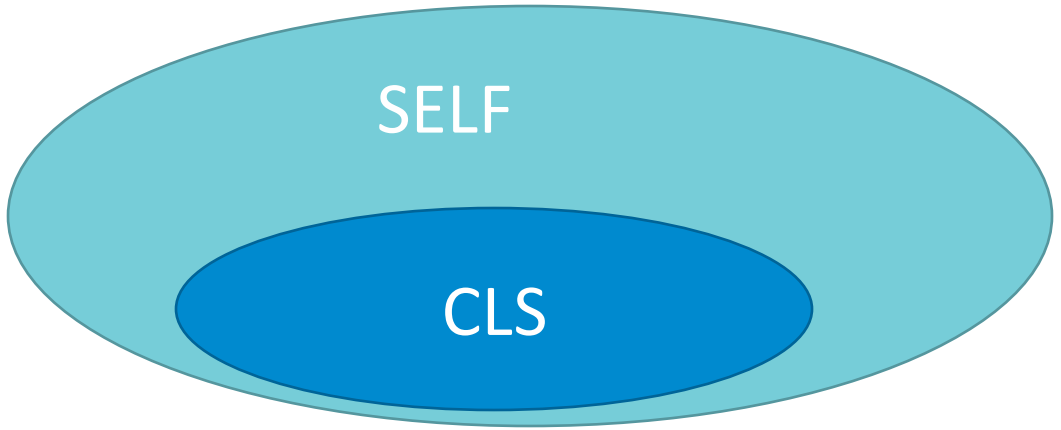
```
class Preson:

    lifespan = 65

    def __init__(self, name):
        self.name = name

    @classmethod
    def increment_lifespan(cls):
        cls.lifespan += 1
```

```
>>> Tom = Person('Thomas')
>>> Marry = Person('Marry')

>>> Tom.lifespan
65
>>> Person.lifespan
65

>>> Person.increment_lifespan()

>>> Person.lifespan
66
>>> Marry.lifespan
66
```

@classmethod decorator

```
class Preson:

    lifespan = 65

    def __init__(self, name):
        self.name = name

    @classmethod
    def increment_lifespan(cls):
        cls.lifespan += 1
```

```
>>> Marry.increment_lifespan()

>>> Tom.lifespan
67
>>> Person.lifespan
67
```



@staticmethod decorator

```
class Dice:

    def __init__(self, number_of_sides):
        self.sides = number_of_sides

    @staticmethod
    def count_outcomes(*dices):
        result = 1
        for item in dices:
            result *= item.sides
        return result
```

```
>>> s = Dice(6)
>>> f = Dice(4)
>>> t = Dice(3)

>>> Dice.count_outcomes(s,f,t)
72

>>> s.count_outcomes(s,f,t)
72
```

@abstractmethod decorator

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def do_something(self):
        pass


class DoStuff(AbstractClassExample):
    pass
```

```
>>> a = DoStuff(228)
TypeError: Can't instantiate
abstract class 'DoStuff' with
abstract methods 'do_something'.
```

@property decorator

```
class SomeClass:
    def __init__(self):
        self._x = 13

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        if type(value) is not int:
            print('Not valid')
        else:
            self._x = value
```

```
>>> obj = SomeClass()

>>> obj.x = 'String'
'Not valid'

>>> obj.x
13
```

Thanks for attention