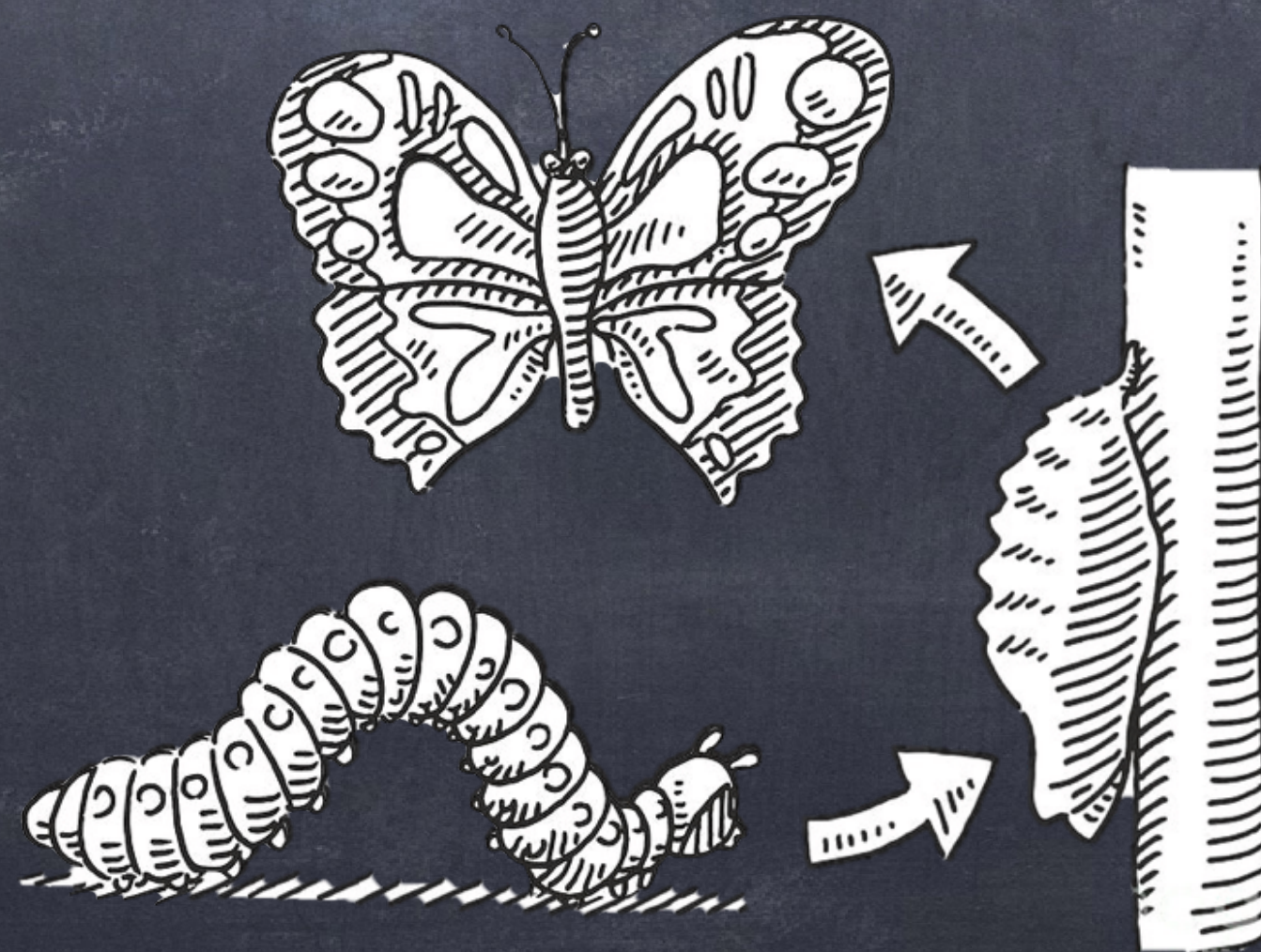


多态



多态 (Polymorphism)

● 多态是指一个实体可以有多种不同的形式 (Polymorphism,

Poly + morphism = multiple forms)

● 主要有三种类型的多态：

◆ 特设多态 (Ad hoc polymorphism)

◆ 参数多态 (Parametric polymorphism)

◆ 子类型多态 (Subtyping or Inclusion polymorphism)

特设多态 (Ad hoc polymorphism)

- 指一个多态函数 (polymorphism functions) 可以应用于有不同类型的实际参数上
- 以来它们所应用到的实际参数类型而有不同的表现。
- 也称为函数重载 (Function Overloading) 或运算符重载 (Operator Overloading)

例子：函数重载

对于Java语言来说：

```
double sum(double a, double b) {
```

```
...
```

```
}
```

Same method name

```
double sum(double a, double b, double c) {
```

```
...
```

```
}
```

different types or numbers of parameters

```
float sum(float a, float b) {
```

```
...
```

```
}
```

调用这些函数时，会在编译时就已经绑定好了具体的函数

例子：函数重载

Python 是动态类型语言，函数调用时，类型推迟到运行时判断，不支持通常的函数重载。

```
def sum(a, b):  
    return a + b
```

```
def sum(a, b, c):  
    return a + b + c
```

```
sum(1.2, 2.2, 1.1)
```

```
sum(1, 2, 3)
```

```
sum(2, 3)
```

Error

例子：函数重载

通过一些手段，可以使Python支持这样的“重载”

```
def add(datatype, *args):  
    if datatype == 'int':  
        ans = 0  
    if datatype == 'str':  
        ans = ''  
    # Traverse through the arguments  
    for i in args:  
        ans = ans + i  
    print(ans)  
# Integer  
add('int', 2, 8)  
# String  
add('str', 'college ', 'Dekho')
```

可变参数

基于类型的分派

Type-based dispatch

例子：函数重载

通过一些手段，可以使Python支持这样的“重载”

缺省参数

```
def multiply(a=None, b=None):  
    if a != None and b == None:  
        print(a)  
    # else will be executed if both are available and returns multiplication of two  
    else:  
        print(a*b)
```

```
# two arguments are passed, returns addition of two  
multiply(2, 12)  
# only one argument is passed, returns a  
multiply(9)
```


例子：函数重载

通过一些手段，可以使Python支持这样的“重载”

```
from multipledispatch import dispatch
```

```
@dispatch(int, int)
```

```
def product(first, second):  
    result = first*second  
    print(result)
```

dispatch修饰符 (Decorator)

```
pip3 install multipledispatch
```

```
@dispatch(int, int, int)  
def product(first, second, third):  
    result = first * second * third  
    print(result)
```

```
# calling product method with 2 arguments  
product(2, 5) # this will give output of 10
```

```
# calling the product method/function with 3 arguments  
product(2, 5, 2) # this will give output of 20
```


例子：运算符重载

- 运算符重载可以拓展运算符可以作用到的操作数上
- 比如：添加实现 `__add__` 方法到自定义的类中，即可完成“+”的重载，使得其可以作用在这个类的对象中

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

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

    def __repr__(self):
        return '{0}, {1}'.format(self.a, self.b)
```

```
c1 = Comb(1, 2)
c2 = Comb(2, 3)
c1 + c2
```


参数多态 (Parametric polymorphism)

- 参数多态允许函数或数据类型被一般性的书写，从而它可以“统一”的处理值而不用依赖于它们的类型。
- 这种函数和数据类型被分别称为“泛化函数” (generic functions) 和“泛化数据类型” (generic datatypes) 从而形成了泛型编程 (generic programming) 的基础。

例子：泛化函数和泛化数据

● 对于Java语言来说：

```
List<String> name = new ArrayList<String>();  
List<Integer> age = new ArrayList<Integer>();  
List<Number> number = new ArrayList<Number>();  
  
void printArray( List<?> data ) {  
    for (i = 0; i < data.length(); i++) {  
        System.out.println(data.get(i));  
    }  
}  
  
printArray(name)  
printArray(age)  
printArray(number)
```


例子：泛化函数和泛化数据

- 对于Python而言，其本身就是动态类型，天然支持泛型

```
def sum_two(a, b):  
    return a + b
```

Anything summable!

- `sum_two`函数对于a和b的类型而言是泛化的

例子：泛化函数和泛化数据

- 实际上，python的这种多态叫做“duck typing”
- 即其并不关心具体类型，而只关注当前方法和属性的集合。



James Whitcomb Riley
"If it looks like a duck and quacks
like a duck, it must be a duck."

例子：泛化函数和泛化数据

```
class Duck:
    def quack(self):
        print("the duck is quacking")
    def feathers(self):
        print("the duck has feathers")
```

```
class Person:
    def quack(self):
        print("the person cannot quack!")
    def feathers(self):
        print("the person has no feathers!")
```

```
donald = Duck()
john = Person()
in_the_forest(donald)
in_the_forest(john)
```

```
def in_the_forest(duck):
    duck.quack()
    duck.feathers()
```

不关心具体类型

只要方法对应即可

例子：泛化函数和泛化数据

● 实际上，python很多内置函数都是这么支持“多态”的

`len()` 作用的对象，只要含有 `__len__()`

`iter()` 作用的对象，只要含有 `__iter__()`

子类型多态

- 它指的是子类型和其父类型的一种可替换关系。
- 意味着在程序中，父类型的所有函数调用，可以被子类型的函数完全替换

例子

在Java中

```
class Animal {
    void eat() {}
}
class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
    public void work() {
        System.out.println("抓老鼠");
    }
}
class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
    public void work() {
        System.out.println("看家");
    }
}
```

```
Animal c = new Cat();
c.eat();
c.work();
Animal d = new Dog();
d.eat();
d.work();
```


例子

在Python中

```
class Parent:
    def f(s):
        print("Parent.f")
    def g(s):
        s.f()
```

```
class Child(Parent):
    def f(me):
        print("Child.f")
```

```
a_child = Child()
a_child.g()
```


特殊方法

- 特殊方法具有内建的行为，其总是以两个下划线开始和结束
- Python通常隐蔽地调用这些方法

Name	Behavior
<code>__init__</code>	Method invoked automatically when an object is constructed
<code>__repr__</code>	Method invoked to display an object as a Python expression
<code>__str__</code>	Method invoked to stringify an object
<code>__add__</code>	Method invoked to add one object to another
<code>__bool__</code>	Method invoked to convert an object to True or False
<code>__float__</code>	Method invoked to convert an object to a float (real number)
<code>__eq__</code>	Method invoked to determine two objects are equal in contents
<code>__gt__</code>	Method invoked to determine whether object is larger than another one

Representation: `__str__` 和 `__repr__`

`__str__`

• `__str__` 方法返回一个对象的可读字符串形式

```
from fractions import Fraction
```

```
one_third = 1/3
```

```
one_half = Fraction(1, 2)
```

```
float.__str__(one_third)           # '0.3333333333333333333333'
```

```
Fraction.__str__(one_half)         # '1/2'
```


str 的用法

● 应用广泛，如 `print()` 函数, `str()` 构造子, f-strings,

```
from fractions import Fraction
```

```
one_third = 1/3
```

```
one_half = Fraction(1, 2)
```

```
print(one_third)
```

```
# '0.33333333333333333333'
```

```
print(one_half)
```

```
# '1/2'
```

```
str(one_third)
```

```
# '0.33333333333333333333'
```

```
str(one_half)
```

```
# '1/2'
```

```
f"{one_half} > {one_third}" # '1/2 > 0.33333333333333333333'
```


自定义 `__str__`

👁️ 重写 `__init__` 方法，来得到一个人类友好的字符串对象

```
class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

    def __init__(self, name):
        self.name = name
    def __str__(self):
        return "Lamb named " + self.name

lil = Lamb("Lil lamb")
str(lil)
print(lil)
```


`__repr__`

- 返回一个字符串，其可以被再次“求值”为对象

```
from fractions import Fraction
```

```
one_half = Fraction(1, 2)
```

```
Fraction.__repr__(one_half)           # 'Fraction(1, 2)'
```

- 如果实现正确，那么调用`eval()`可以得到一个值相同的对象

```
another_half = eval(Fraction.__repr__(one_half))
```


`__repr__` 的用法

- 一般用于 `repr(object)` 调用，以及在python的交互式终端显示对象

```
from fractions import Fraction
```

```
one_third = 1/3
```

```
one_half = Fraction(1, 2)
```

```
one_third
```

```
one_half
```

```
repr(one_third)
```

```
repr(one_half)
```


自定义 `__repr__`

● 写 `__repr__` 方法，来得到一个合适的Python对象表示

```
class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

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

    def __str__(self):
        return "Lamb named " + self.name

    def __repr__(self):
        return f"Lamb({repr(self.name)})"

lil = Lamb("Lil lamb")
repr(lil)
lil
```


`__repr__` 和 `__str__` 的一些规则

① 当 `repr(obj)` 函数被调用时

- ① Python 调用 `ClassName.__repr__` 方法 (如果存在)

- ② 如果找不到, 会在父类上找相应的 `__repr__`

- ③ 如果都失败了, 调用 `object.__repr__`

① 当 `str(obj)` 函数被调用时

- ① Python 调用 `ClassName.__str__` 方法 (如果存在)

- ② 如果找不到, 调用 `__repr__`

- ③ 之后如上

Any questions ?