Python编程基础

1 编程的典型场景 — 数据处理

编程的典型场景

- 数据处理
  - 输入
  - 处理
  - 输出

求两个数和

- 输入
  - 控制台输入2个数
- 处理
  - 将输入String转出int
  - 求和
- 输出
  - 输出和

2 编程基本概念 — 分支（选择）

决策

- 选择场景
  - 布尔条件
    - 如果 n是奇数时。。。； n是偶数时。。。
  - 可哈希的值
    - 键值对、字典

3 编程基本概念 — 重复（循环）

典型场景

- 计数器
  - 提示用户输入5个数字，计算他们的和
- 直到条件满足
  - 当用户输入键入无效的输入值，不要退出，而是不断提示用户输入，直到获得有效的输入值
- 递归
  - 通过递归实现循环
- 嵌套循环
  - 乘法表

循环计数与循环语句

- 循环计数与循环内容无关
  - 打印HelloWorld 5次
- 循环计数与循环内容有关
  - 计算1+2+。。。+100的和

编程基本概念 — 流程图

流程图

开始

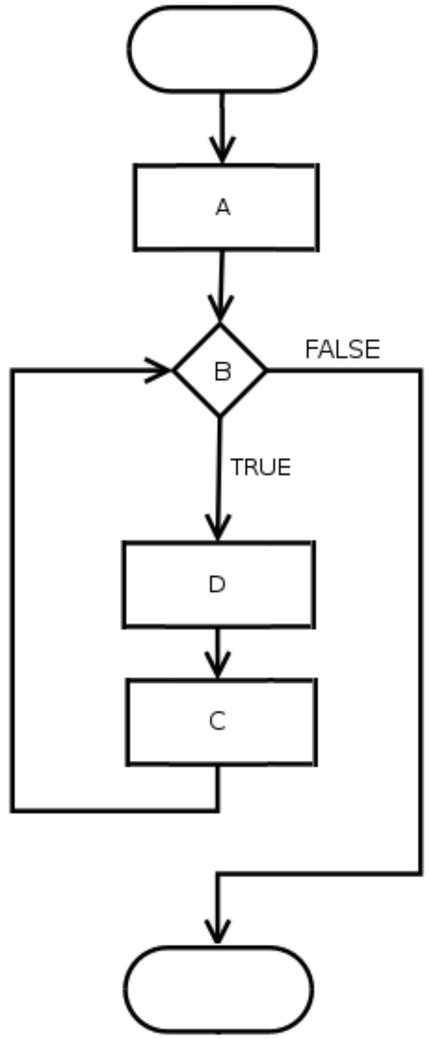
结束

操作

条件

流

```
for(A;B;C)
  D;
```



4 编程基本概念 — 方法（函数）

为什么要方法调用?

如果没有方法调用?

方法（函数）的意义

- 逻辑的封装
- 重用
- 可修改

方法被调用

- 方法被调用的特性
  - 每个方法都只有一个入口。
  - 当执行被调用的方法的时候，调用方法暂停。
  - 当方法结束时，程序的控制权交还给调用处。

声明与调用

```
def say_hello():
    #block in a function
    print('Hello world')
say_hello()
say_hello()
```

形参（parameter）与实参（argument）

- 形参
  - A parameter is the variable which is part of the method’s signature (method declaration).
- 实参
  - An argument is an expression used when calling the method.

例子

```
def print_max(a,b):
    if a>b:
        print(a,'is maximum')
    elif a == b:
        print(a, 'is equal to',b)
    else:
        print(b,'is maximum' )


print_max(3,4)


x = 5
y = 7


print_max(x,y)
```

多参数

- 一一对应
  - f(int a, int b)
  - f(2,3)

必须参数

- #!/usr/bin/python3
-
- #可写函数说明
- def printme(str):
- "打印任何传入的字符串"
- print (str);
- return;
-
- #调用printme函数
- printme();
-
- 以上实例输出结果：
- Traceback (most recent call last):
- File "test.py", line 10, in <module>
- printme();
- TypeError: printme() missing 1 required positional argument: 'str'
-

关键字参数

- #!/usr/bin/python3
-
- #可写函数说明
- def printme(str):
- "打印任何传入的字符串"
- print (str);
- return;
-
- #调用printme函数
- printme(str = "菜鸟教程");
- 以上实例输出结果：
- 菜鸟教程

不需要指定顺序

- #!/usr/bin/python3
-
- #可写函数说明
- def printinfo(name, age):
- "打印任何传入的字符串"
- print ("名字: ", name);
- print ("年龄: ", age);
- return;
-
- #调用printinfo函数
- printinfo(age=50, name="runoob");

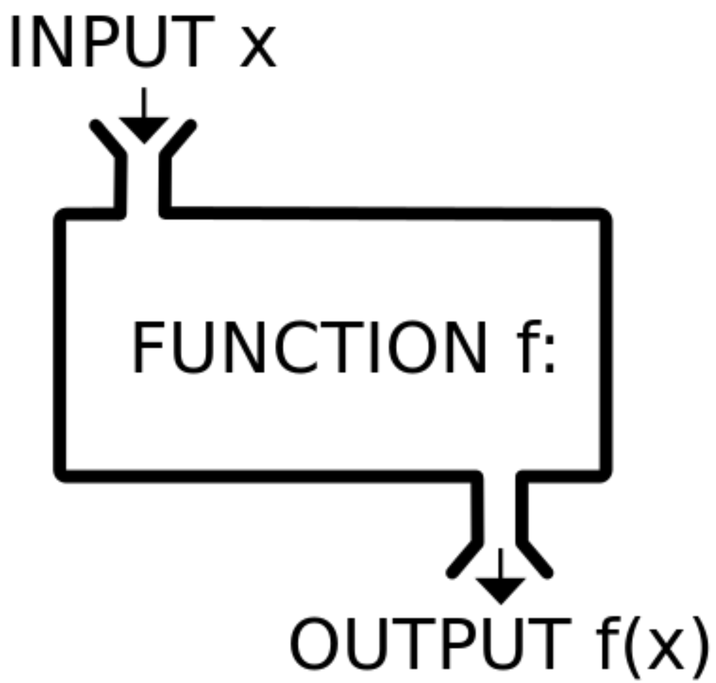
- 以上实例输出结果：
- 名字：runoob
- 年龄：50

默认参数

- #!/usr/bin/python3
-
- #可写函数说明
- def printinfo(name, age = 35):
- "打印任何传入的字符串"
- print ("名字: ", name);
- print ("年龄: ", age);
- return;
-
- #调用printinfo函数
- printinfo(age=50, name="runoob");
- print ("------------------------")
- printinfo(name="runoob");

- 以上实例输出结果：
- 名字: runoob
- 年龄: 50
- ------------------------
- 名字: runoob
- 年龄: 35

返回值



返回值 例子

- #!/usr/bin/python3
- # 可写函数说明
- def sum(arg1, arg2):
- # 返回2个参数的和."
- total = arg1 + arg2
- print ("函数内 : ", total)
- return total;
- # 调用sum函数
- total = sum(10, 20);
- print ("函数外 : ", total)
- 以上实例输出结果：
- 函数内： 30
- 函数外： 30

多返回值

- >>> def myfun():
- ... return 1, 2, 3
- ...
- >>> a, b, c = myfun()
- >>> a
- 1
- >>> b
- 2
- >>> c
- 3
-

main函数

- 程序的入口点
- #hello.py
- def foo():
- 	str="function"
- 	print(str);
- if __name__==“__main__”：#如果没有这个main函数，运行python无结果
- 	print("main")
- 	foo()

- 运行这个python文件，输出结果
- main
- function

5 编程基本概念 — 测试

有代码就得有测试！

黑盒测试

黑盒测试

- 不了解程序的内部情况
- 只知道程序的输入、输出和系统的功能

黑盒测试一般流程

- 0. 初始化测试
- 1. 调用被测方法得到实际结果result
- 2. 与期望的结果相比较
- 3. 输出测试的结果

源代码

- """write your code in method currency_exchange"""

- def currency_exchange():
- euro = int(input("How many euros are you exchanging?"))
- rate = float((input("What is the exchange rate?")))
- dollar = round(euro*rate/100, 2)
- print("{0} euros at an exchange rate of {1} is {2} U.S. dollars.".format(euro, rate, dollar))

测试代码

```
class TestCurrency(unittest.TestCase):
    """test currency exchange"""

    def setUp(self):
        self.stream_out = MyStream()
        self.stream_in = MyStream()
        self.out_stream = sys.stdout
        self.in_stream = sys.stdin
        sys.stdout = self.stream_out
        sys.stdin = self.stream_in
        pass

    def test_currency1(self):
        euro = '97'
        rate = '31'
        self.stream_in.write(euro)
        self.stream_in.write(rate)
        currency.currency_exchange()
        result = float(str(self.stream_out.buff[2]).split(" ")[9])
        self.assertEqual(result, 30.07)
```

成功

- ----------
- Ran 4 tests in 0.015s

- OK

修改下源代码

- """write your code in method currency_exchange"""

- def currency_exchange():
- euro = int(input("How many euros are you exchanging?"))
- rate = float((input("What is the exchange rate?")))
- dollar = euro*rate/100
- print("{0} euros at an exchange rate of {1} is {2} U.S. dollars.".format(euro, rate, dollar))

错误提示

- 111.38 != 111.3831

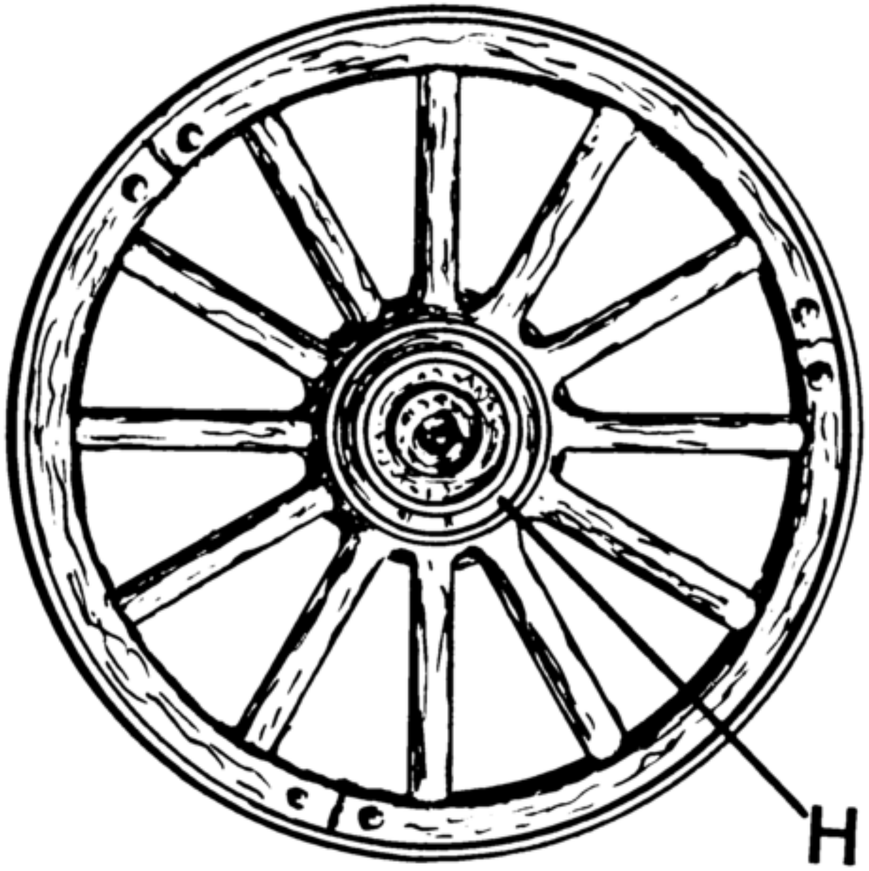
- Expected :111.3831
- Actual   :111.38
- <Click to see difference>

- Traceback (most recent call last):
- File "/Applications/PyCharm CE.app/Contents/helpers/pycharm/teamcity/diff_tools.py", line 30, in _patched_equals
- old(self, first, second, msg)
- File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/case.py", line 821, in assertEqual
- assertion_func(first, second, msg=msg)
- File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/case.py", line 814, in _baseAssertEqual
- raise self.failureException(msg)
- AssertionError: 111.3831 != 111.38

- During handling of the above exception, another exception occurred:

- Traceback (most recent call last):
- File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
- yield
- File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/case.py", line 601, in run
- testMethod()
- File "/Users/liuqin/Downloads/homework2 2/test/currency_test.py", line 43, in test_currency3
- self.assertEqual(result, 111.38)

6 模块

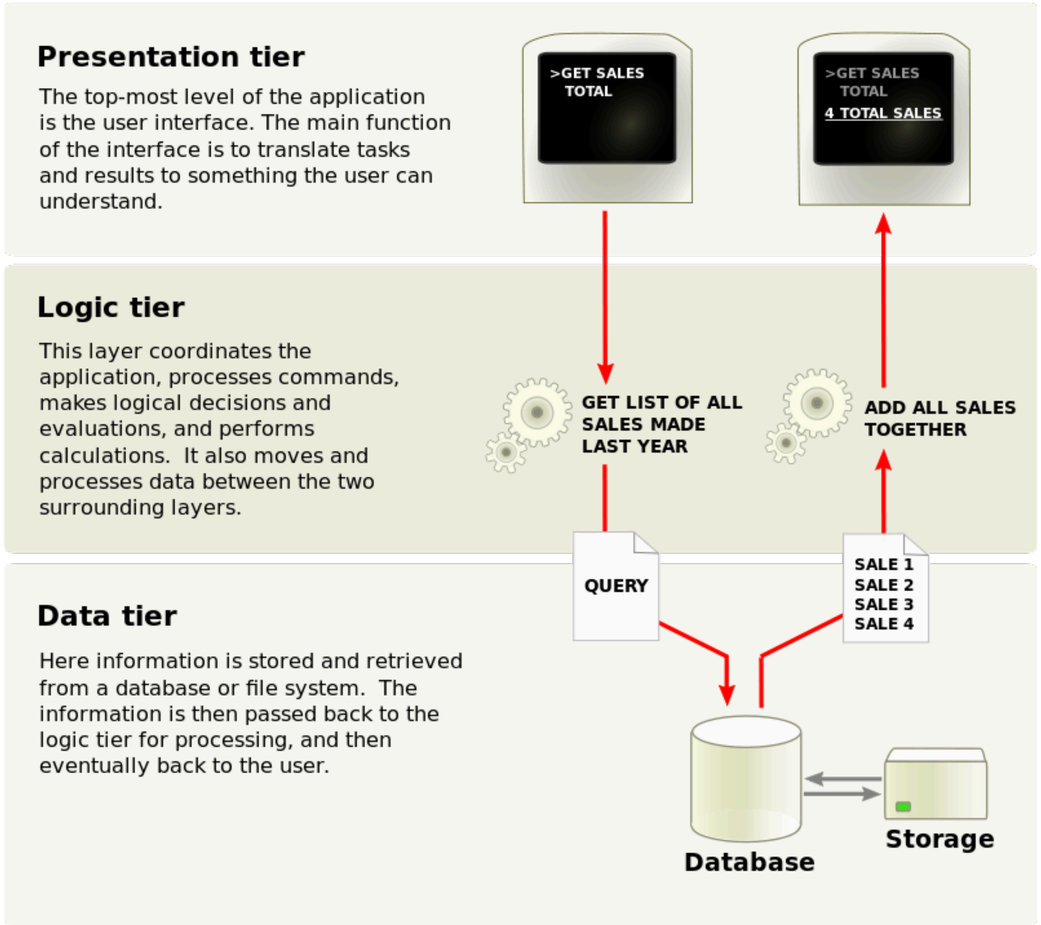


人类历史上一项伟大的发明

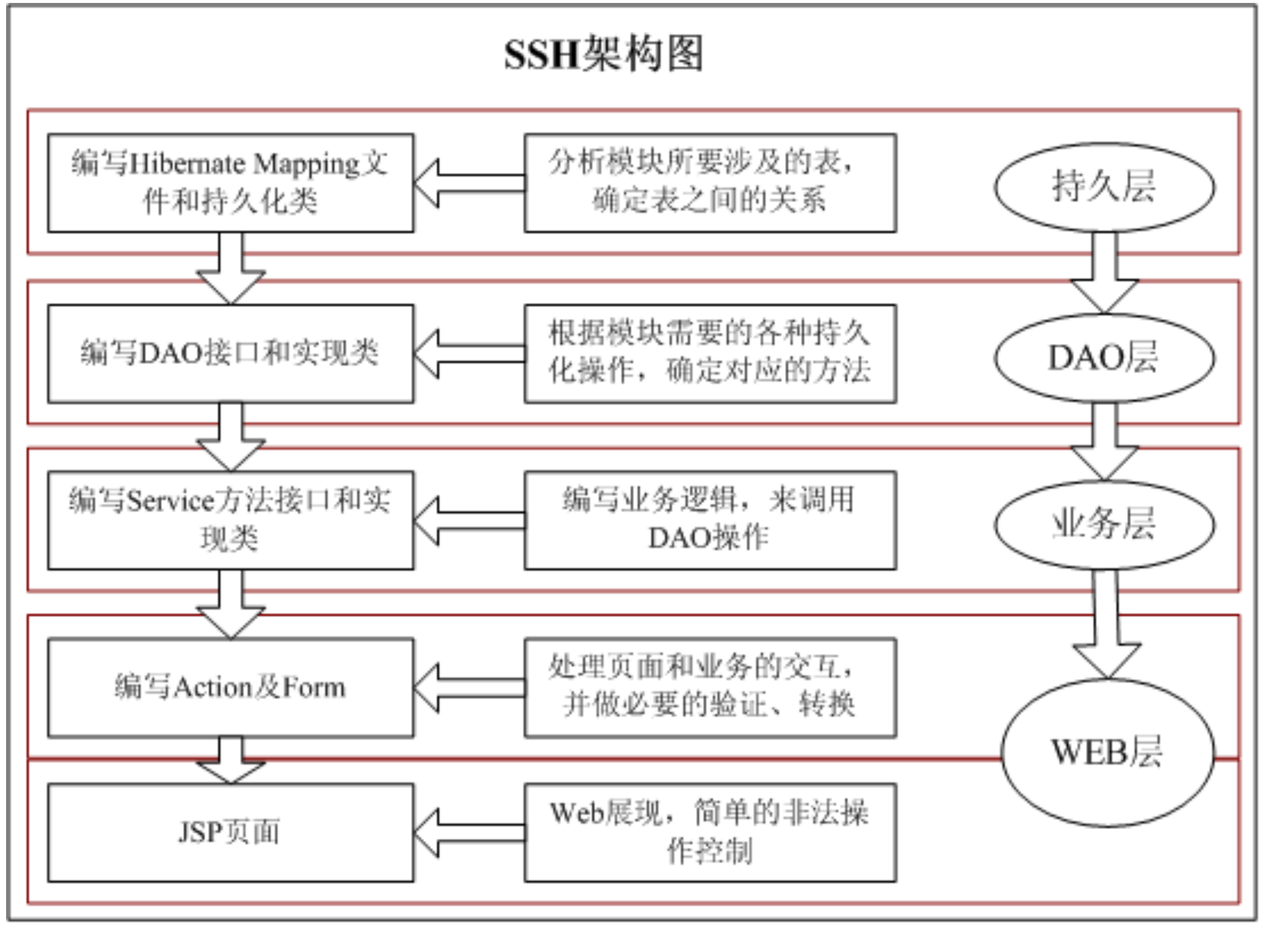
不要重新发明轮子

重用

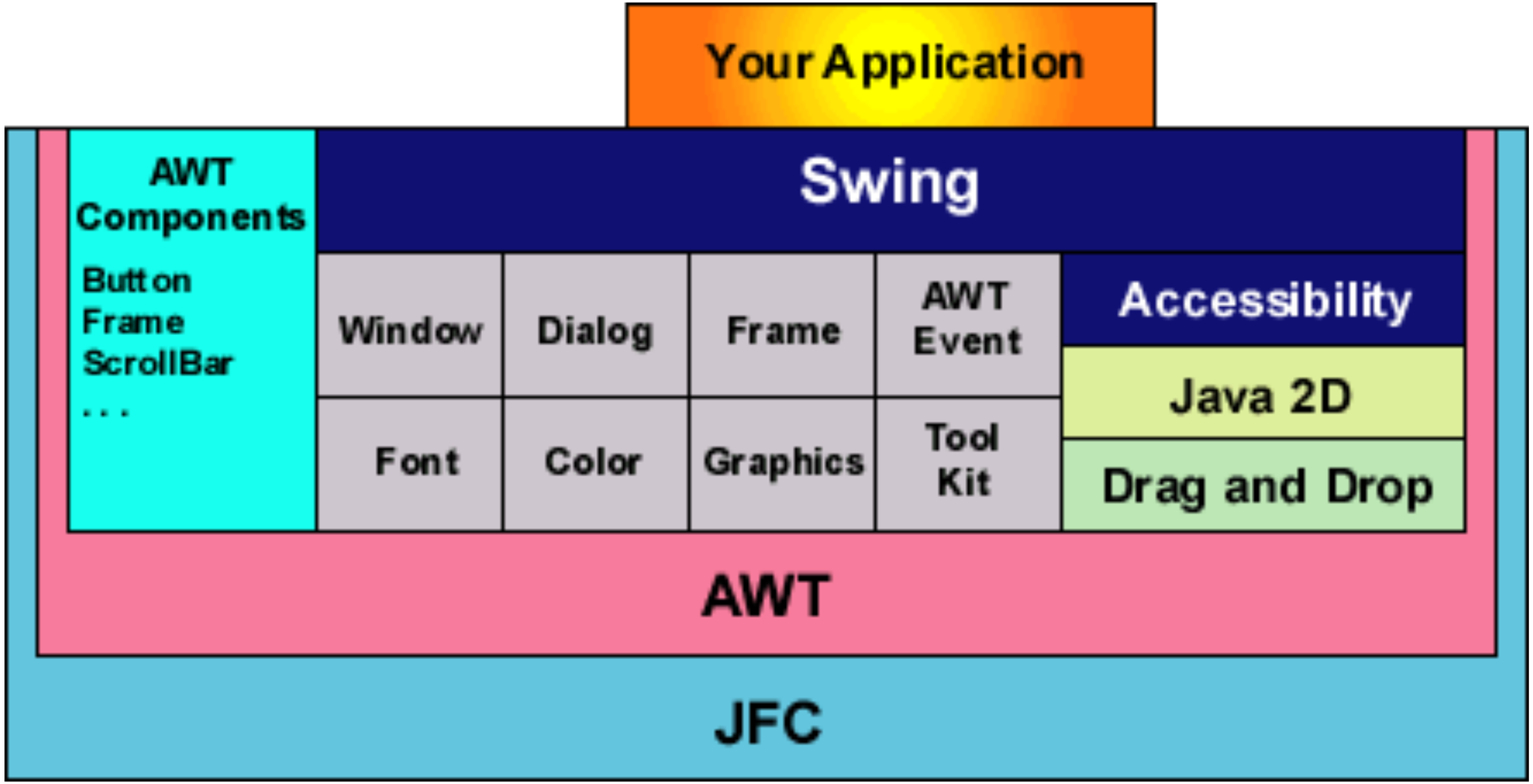
- 体系结构风格
  - 分层风格
- 框架
  - SSH
- 库
  - SWING、AWT
- 设计模式
  - 策略模式
- 代码重用
  - 类、方法


Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
>GET SALES TOTAL
>GET SALES TOTAL
4 TOTAL SALES
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
QUERY
SALE 1
SALE 2
SALE 3
SALE 4
Database
Storage

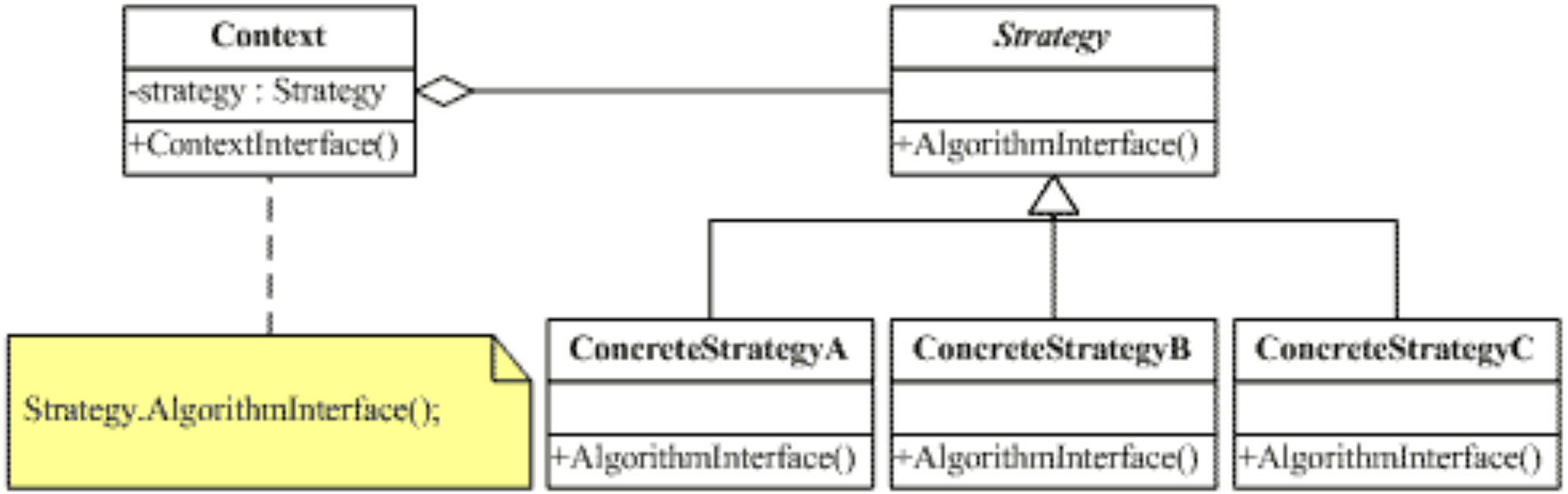
分层风格


SSH架构图
编写Hibernate Mapping文件和持久化类
分析模块所要涉及的表，确定表之间的关系
持久层
编写DAO接口和实现类
根据模块需要的各种持久化操作，确定对应的方法
DAO层
编写Service方法接口和实现类
编写业务逻辑，来调用DAO操作
业务层
编写Action及Form
处理页面和业务的交互，并做必要的验证、转换
WEB层
JSP页面
Web展现，简单的非法操作控制

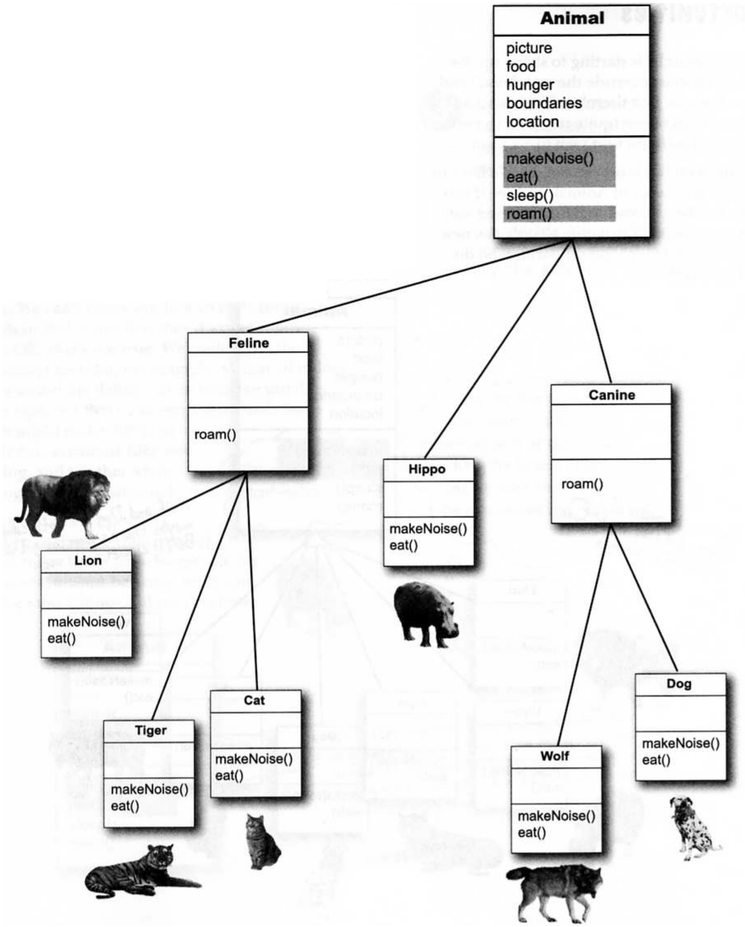
SSH框架


Your Application
AWT Components
Button
Frame
ScrollBar
. . .
Swing
Window
Dialog
Frame
AWT Event
Accessibility
Java 2D
Font
Color
Graphics
Tool Kit
Drag and Drop
AWT
JFC

Swing AWT 库


Context
-strategy : Strategy
+ContextInterface()
Strategy
+AlgorithmInterface()
Strategy.AlgorithmInterface();
ConcreteStrategyA
+AlgorithmInterface()
ConcreteStrategyB
+AlgorithmInterface()
ConcreteStrategyC
+AlgorithmInterface()

策略模式


Animal
picture
food
hunger
boundaries
location
makeNoise()
eat()
sleep()
roam()
Feline
roam()
Canine
roam()
Hippo
makeNoise()
eat()
Lion
makeNoise()
eat()
Tiger
makeNoise()
eat()
Cat
makeNoise()
eat()
Wolf
makeNoise()
eat()
Dog
makeNoise()
eat()

继承

7 文件

持久化

持久化

- 文件
- 对象序列化
- 网络

CSV文件

- 逗号分隔值（Comma-Separated Values，CSV，有时也称为字符分隔值，因为分隔字符也可以不是逗号），其文件以纯文本形式存储表格数据（数字和文本）。纯文本意味着该文件是一个字符序列，不含必须象二进制数字那样被解读的数据。CSV文件由任意数目的记录组成，记录间以某种换行符分隔；每条记录由字段组成，字段间的分隔符是其它字符或字符串，最常见的是逗号或制表符。通常，所有记录都有完全相同的字段序列。

- CSV文件格式的通用标准并不存在，但是在RFC 4180中有基础性的描述。使用的字符编码同样没有被指定，但是7-bit ASCII是最基本的通用编码。

样例

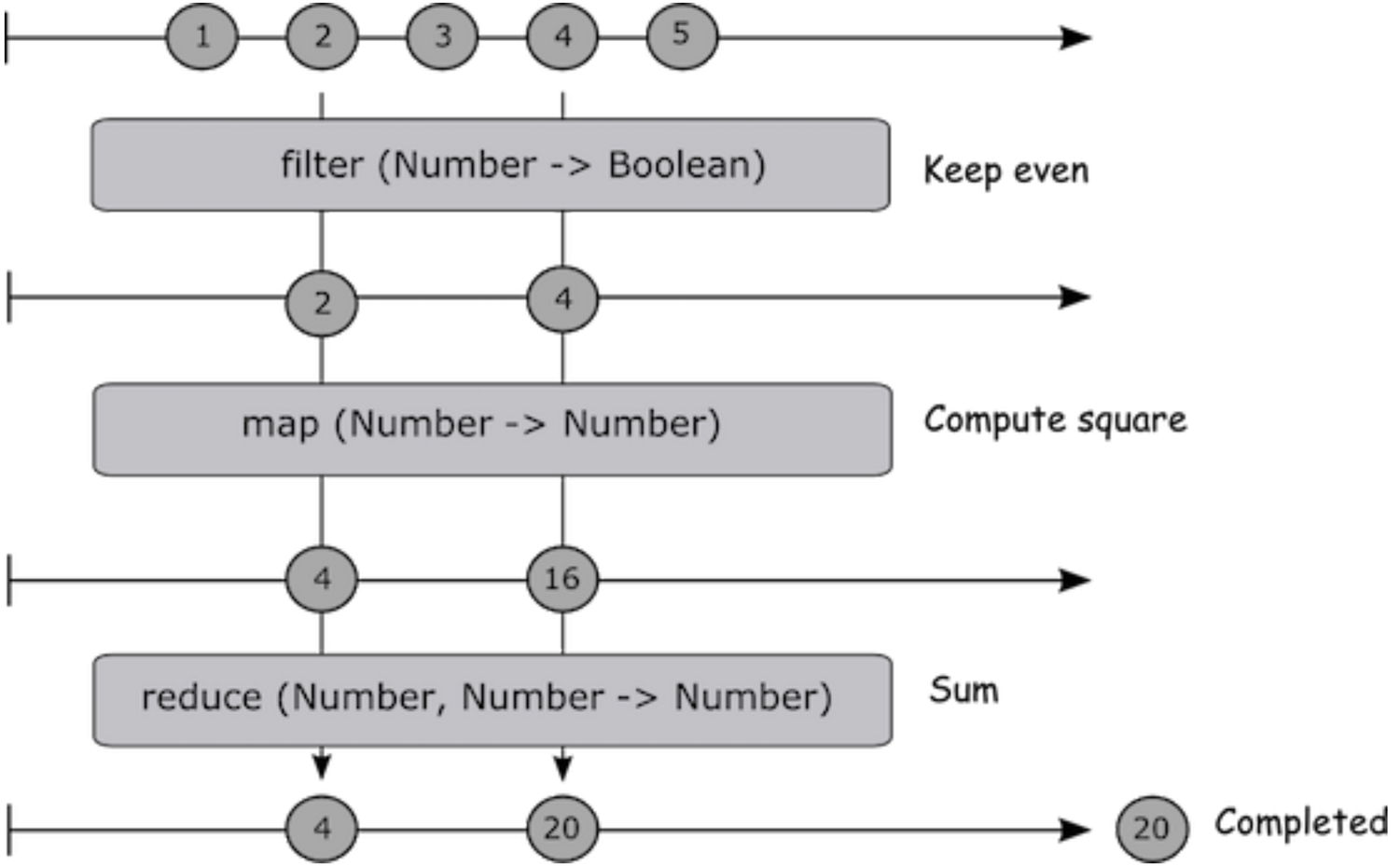
- Year,Make,Model,Description,Price
- 1997,Ford,E350,"ac, abs, moon",3000.00
- 1999,Chevy,"Venture ""Extended Edition""","",4900.00
- 1999,Chevy,"Venture ""Extended Edition, Very Large""",,5000.00
- 1996,Jeep,Grand Cherokee,"MUST SELL!
- air, moon roof, loaded",4799.00

年份	品牌	型号	描述	价格
1997	Ford	E350	ac, abs, moon	3000.00
1999	Chevy	Venture "Extended Edition"		4900.00
1999	Chevy	Venture "Extended Edition, Very Large"		5000.00
1996	Jeep	Grand Cherokee	MUST SELL! air, moon roof, loaded	4799.00

8 函数式编程

Python语言函数式编程

- 基本函数：map()、reduce()、filter()
- 算子(operator)：lambda


1
2
3
4
5
filter (Number -> Boolean)
Keep even
2
4
map (Number -> Number)
Compute square
4
16
reduce (Number, Number -> Number)
Sum
4
20
20
Completed

Lambdar

- func1 = lambda : <expression()>
- func2 = lambda x : <expression(x)>
- func3 = lambda x,y : <expression(x,y)>

- >>> func = lambda : 123
- >>> func
- <function <lambda> at 0x100f4e1b8>
- >>> func()
- 123

Map

- map(func, iterable)

- >>> double_func = lambda s : s * 2
- >>> map(double_func, [1,2,3,4,5])
- [2, 4, 6, 8, 10]

- map(func, iterable1, iterable2)

- >>> plus = lambda x,y : (x or 0) + (y or 0)
- >>> map(plus, [1,2,3], [4,5,6])
- [5, 7, 9]
- >>> map(plus, [1,2,3,4], [4,5,6])
- [5, 7, 9, 4]
- >>> map(plus, [1,2,3], [4,5,6,7])
- [5, 7, 9, 7]

Reduce

- reduce(func, iterable[, initializer])

- >>> plus = lambda x, y : x + y
- >>> reduce(plus, [1,2,3,4,5])
- 15
- >>> reduce(plus, [1,2,3,4,5], 10)
- 25

Filter

- filter(func, iterable)

- >>> mode2 = lambda x : x % 2
- >>> filter(mode2, [1,2,3,4,5,6,7,8,9,10])
- [1, 3, 5, 7, 9]

替换条件控制语句

- # flow control statement
- if <cond1>: func1()
- elif <cond2>: func2()
- else: func3()

- # Equivalent "short circuit" expression
- (<cond1> and func1()) or (<cond2> and func2()) or (func3())

- >>> pr = lambda s:s
- >>> print_num = lambda x: (x==1 and pr("one")) \
- .... or (x==2 and pr("two")) \
- .... or (pr("other"))
- >>> print_num(1)
- 'one'
- >>> print_num(2)
- 'two'
- >>> print_num(3)
- 'other'

替换循环控制语句

- # statement-based for loop
- for e in lst: func(e)

- # Equivalent map()-based loop
- map(func, lst)

- >>> square = lambda x : x * x
- >>> for x in [1,2,3,4,5]: square(x)

- ...
- 1
- 4
- 9
- 16
- 25

- >>> map(square, [1,2,3,4,5])
- [1, 4, 9, 16, 25]

- # statement-based while loop
- while <condition>:
- <pre-suite>
- if <break_condition>:
- break
- else:
- <suite>

- # Equivalent FP-style recursive while loop
- def while_block():
- <pre-suite>
- if <break_condition>:
- return 1
- else:
- <suite>
- return 0
-
- while_FP = lambda: <condition> and (while_block() or while_FP())
- while_FP()

示例 — echo

```
# imperative version of "echo()"
def echo_IMP():
    while 1:
        x = raw_input("IMP -- ")
        print x
        if x == 'quit':
            break

echo_IMP()
```

```
def monadic_print(x):
    print x
    return x

# FP version of "echo()"
echo_FP = lambda: monadic_print(raw_input("FP -- "))=='quit' or echo_FP()

echo_FP()
```

示例 — 综合

- 笛卡尔积元组集合中两个元素之积大于25的所有元组。

- from functools import reduce

- bigmuls = lambda xs,ys: filter(lambda x,y:x*y > 25, combine(xs,ys))
- combine = lambda xs,ys: map(None, xs*len(ys), dupelms(ys,len(xs)))
- dupelms = lambda lst,n: reduce(lambda s,t:s+t, map(lambda l,n=n: [l]*n, lst))

- print bigmuls([1,2,3,4],[10,15,3,22])

- [(3, 10), (4, 10), (2, 15), (3, 15), (4, 15), (2, 22), (3, 22), (4, 22)]

简单例子

[foo.py]

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

[demo.py]

```
import foo

a = [1, 'python']
a = 'a string'

def func():
a = 1
b = 257
print(a + b)

print(a)

if __name__ == '__main__':
func()
foo.add(1, 2)
```

执行这个程序

```
python demo.py
```

输出结果

```
a string
258
```

编译

执行 `python demo.py` 后，将会启动 Python 的解释器，然后将 demo.py 编译成一个字节码对象 PyCodeObject。

在 Python 的世界中，一切都是对象，函数也是对象，类型也是对象，类也是对象（类属于自定义的类型，在 Python 2.2 之前，int, dict 这些内置类型与类是存在不同的，在之后才统一起来，全部继承自 object），甚至连编译出来的字节码也是对象，.pyc 文件是字节码对象（PyCodeObject）在硬盘上的表现形式。

dis

Python 标准库提供了用来生成代码对应字节码的工具 `dis`。dis 提供一个名为 dis 的方法，这个方法接收一个 code 对象，然后会输出 code 对象里的字节码指令信息。

```
s = open('demo.py').read()
co = compile(s, 'demo.py', 'exec')
import dis
dis.dis(co)
```

执行上面这段代码可以输出 demo.py 编译后的字节码指令

```
1 0 LOAD_CONST 0 (-1)
3 LOAD_CONST 1 (None)
6 IMPORT_NAME 0 (foo)
9 STORE_NAME 0 (foo)

3 12 LOAD_CONST 2 (1)
15 LOAD_CONST 3 (u'python')
18 BUILD_LIST 2
21 STORE_NAME 1 (a)

4 24 LOAD_CONST 4 (u'a string')
27 STORE_NAME 1 (a)

6 30 LOAD_CONST 5 (<code object func at 00D97650, file
"demo.py", line 6>)
33 MAKE_FUNCTION 0
36 STORE_NAME 2 (func)

11 39 LOAD_NAME 1 (a)
42 PRINT_ITEM
43 PRINT_NEWLINE

13 44 LOAD_NAME 3 (__name__)
47 LOAD_CONST 6 (u'__main__')
50 COMPARE_OP 2 (==)
53 POP_JUMP_IF_FALSE 82

14 56 LOAD_NAME 2 (func)
59 CALL_FUNCTION 0
62 POP_TOP

15 63 LOAD_NAME 0 (foo)
66 LOAD_ATTR 4 (add)
69 LOAD_CONST 2 (1)
72 LOAD_CONST 7 (2)
75 CALL_FUNCTION 2
78 POP_TOP
79 JUMP_FORWARD 0 (to 82)
>> 82 LOAD_CONST 1 (None)
85 RETURN_VALUE
```

执行

demo.py 被编译后，接下来的工作就交由 Python 虚拟机来执行字节码指令了。Python 虚拟机会从编译得到的 PyCodeObject 对象中依次读入每一条字节码指令，并在当前的上下文环境中执行这条字节码指令。我们的程序就是通过这样循环往复的过程才得以执行。

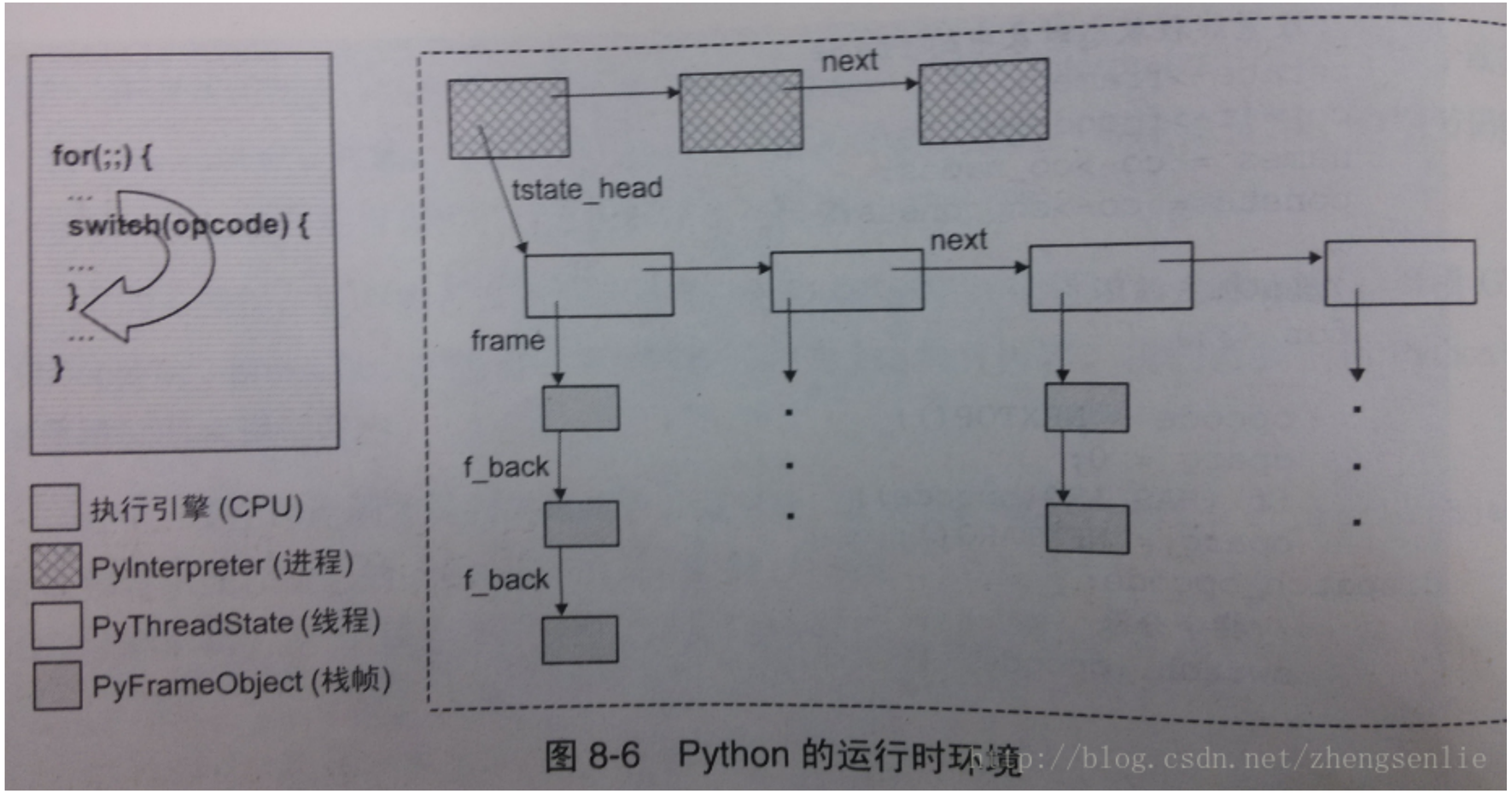


图 8-6　Python 的运行时环境

运行环境