Sython设计规格说明书

1. 语言设计的背景及范型

【内容概要】需说明语言设计的动机以及语言的特点。

* 1. 介绍

Sython取名来源于Simple Python，即为简单的Python语言，它是一门无类型、面向对象的脚本语言，在基本语法层面上，较为类似C语言。同时，它是解释型语言，即Sython解释器将源代码转换为语法树，然后再由解释器分析语法树并执行。

本小组从C语言的基本文法入手，通过自己对Python语言语法规则的理解以及一些其他资料的学习，以C文法为基础，设计了Sython文法，既支持面向对象的抽象编程也支持面向过程的函数编程。通过对lex&yacc的学习，本小组为Sython设计制作了词法语法分析器，并部分完成了Sython的解释器。

* 1. 设计动机

目前，Python的使用热度逐步上升，其特点是“优雅”、“明确”、“简单”，其最终的代码不但看起来简单美观，并且功能同样十分强大。初学者学Python，相比于学习C\C++等语言简单了很多。然而，正是由于其简单性，我们在学习Python的时候，很多时候都是倾听点水，浅尝辄止。同时，Python语言也有一些让人又爱又恨的特性，例如不需要花括号，只需要缩进，不需要分号，只需要换行。这样确实使得代码看起来简洁、美观，但是有时候一个小空格造成的小bug确困扰人很久。

本小组为加深对Python语言的了解以及深入学习程序语言的设计，从C语言出发，按照我们的设计，将其扩充为一种类似Python语言的脚本语言，Sython。相比于c语言,Sython更简单易学，上手快，同时也具有可扩展性和可嵌入型。我们在Sython中引入了类，从而Sython既支持面向对象的抽象编程也支持面向过程的函数编程。相比与Python，Sython当然比不了，但是我们按照自己意图定制了一门语言，并拥有大部分Python的特性，这已经达到了我们的意图。

我们为Sython设计了解释器，在一步步的实现中，也加深了自己对于编译原理的理解以及脚本语言的运行方式。

1. 语言的语法、语义规格说明

【内容概要】需说明语言的控制语句、类型系统、函数、复合类型变量的语法和语义设计。

2.1 语言词法和语法设计

【内容概要】需采用EBNF产生式来描述语言的词法和语法。

2.1.1 终结符

| **string** | **终结符** | **string** | **终结符** |
| --- | --- | --- | --- |
| "(" | LP | "<" | LT |
| ")" | RP | "<=" | LE |
| "[" | LB | "+=" | ADD\_ASSIGN\_T |
| "]" | RB | "-=" | SUB\_ASSIGN\_T |
| "{" | LC | "\*=" | MUL\_ASSIGN\_T |
| "}" | RC | "/=" | DIV\_ASSIGN\_T |
| ";" | SEMICOLON | "%=" | MOD\_ASSIGN\_T |
| "," | COMMA | "!" | EXCLAMATION |
| "&&" | LOGICAL\_AND | "~" | '~' |
| "||" | LOGICAL\_OR | "+" | ADD |
| "=" | ASSIGN\_T | "++" | INCREMENT |
| "==" | EQ | "--" | DECREMENT |
| "!=" | NE | "-" | SUB |
| ">=" | GE | "\*" | MUL |
| "/" | DIV | "%" | MOD |
| "." | DOT | ">" | GT |

2.1.2 EBNF产生式

| **非终结符** | **产生式** |
| --- | --- |
| program | ::= {translation\_unit} |
| translation\_unit | :: = {definition\_or\_statement} definition\_or\_statement |
| definition\_or\_statement | :: =function\_definition| statement| def\_class| def\_calss |
| def\_class | :: ="class" IDENTIFIER class\_extend "{" field\_def methods\_def "}" |
| class\_extend | :: = {"extend" IDENTIFIER} |
| field\_def | :: = ["static"] instant\_filed |
| instant\_filed | :: = expression ";" |
| methods\_def | :: = method\_or\_gs |
| method\_or\_gs | :: ={method\_def | getter\_def | setter\_def} |
| method\_def | :: = ["static"] IDENTIFIER "(" [parameter\_list] ")" block |
| getter\_def | :: = ["static"] IDENTIFIER block |
| setter\_def | :: = ["static"] "IDENTIFIER" "=" "(" block ")" block |
| function\_definition | :: = "function" IDENTIFIER "(" [parameter\_list] ")" block |
| parameter\_list | :: = [parameter\_list ","] IDENTIFIER |
| argument\_list | :: = [argument\_list ","] assignment\_expression |
| expression | :: = [expression ","] assignment\_expression |
| assignment\_expression | ::={["FINAL"] postfix\_expression "=" | "+=" | "-=" | "\*=" | "/=" | "%="} logical\_or\_expression |
| logical\_or\_expression | ::= {lgical\_and\_expression "||" } logical\_and\_expression |
| logical\_and\_expression | ::={equality\_expression "&&" } equality\_expression |
| equality\_expression | ::={relation\_expression "==" | "!=" }relation\_expression |
| relation\_expression | ::={additive\_expression ">" | ">=" | "<="} additive\_expression |
| additive\_expression | ::= {multicative\_expression "+" | "-" } multiplicative\_expression |
| multiplicative\_expression | ::={unary\_expression "\*" | "/" | "%"} unary\_expression |
| unary\_expression | ::={"-" | "!" }postfix\_expression |
| postfix\_expression | :: = {postfix\_expression ("[" expression "]" | ( "." IDENTIFIER) | ( "(" [argument\_list] ")") | "++" | "--"} primary\_expression |
| primary\_expression | ::= "[" expression "]" | IDENTIFIER | INT\_LITERAL | DOUBLE | STRING | TRUE | FALSE | NULL | array\_literal | closure\_defination |
| array\_literal | ::= "{" expression\_list [","] "}" |
| closure\_defination | ::= "closure" [IDENTIFIER] "(" [parameter\_list] ")" block |
| expression\_list | ::= {expression\_list ","} [assignment\_expression] |
| statement | expression ";" | global\_statement | if\_statement | while\_statement | for\_statement | foreach\_statement | return\_statement | break\_statement | continue\_statement | try\_statement | throw\_statement |
| global\_statement | ::= "global" identifier\_list |
| identifier\_list | ::= {identifier\_list ","} IDENTIFIER |
| if\_statement | :: ="if" "(" expression ")" block [elsif\_list] ["else" block] |
| elsif\_list | ::= {elsif\_list} elsif |
| elsif | ::= "elsif" "(" expression ")" block |
| while\_statement | ::= "while" "(" expression ")" block |
| for\_statement | ::= "for" "(" [expression] ";" [expression] ";" [expression] ")" block |
| foreach\_statement | ::= "foreach" "(" IDENTIFIER ":" expression ")" block |
| return\_statement | ::= "return " [expression] ";" |
| break\_statement | :: = "break" ";" |
| continue\_statement | :: = "continue" ";" |
| try\_statement | ::= "try" block ["catch" "(" IDENTIFIER ")" block] ["finally" block] |
| throw\_statement | ::= "throw" expression ";" |
| block | ::= "{" [statement\_list] "}" |

2.2 语言语义设计

【内容概要】需采用指称语义来描述语言的词法和语法。

**2.2.1 值域**

Value = truth\_value Truth\_Value + integer Integer + double Double + string String + array\_value Array\_Value

值域相关的辅助函数（在函数域中）：

sum: Value × Value → Value

difference: Value × Value → Value

mult: Value × Value → Value

div: Value × Value → Value

mod: Value × Value → Value

negat: Value × Value → Value

isequ: Value × Value → Value

or: Value × Value → Value

and: Value × Value → Value

bgt: Value × Value → Value

bge: Value × Value → Value

blt: Value × Value → Value

ble: Value × Value → Value

以上辅助函数形式化定义：

sum(val1, val2) = if abs(val1+val2) <= maxint

then val1 + val2

else ⊥

sum(⊥, val2) = ⊥

sum(val1, ⊥) = ⊥

difference(val1, val2) = if abs(val1-val2) <= maxint

then val1 - val2

else ⊥

difference (⊥, val2) = ⊥

difference (val1, ⊥) = ⊥

mult(val1, val2) = if abs(val1 \* val2) <= maxint

then val1 \* val2

else ⊥

mult (⊥, val2) = ⊥

mult (val1, ⊥) = ⊥

div(val1, val2) = value val =

if abs(val1 / val2) <= maxint or val2 == 0

then val1 / val2

else ⊥

div (⊥, val2) = ⊥

div (val1, ⊥) = ⊥

mod(val1, val2) =

if abs(val1 % val2) <= maxint or val2 == 0

then val1 % val2

else ⊥

mod (⊥, val2) = ⊥

mod (val1, ⊥) = ⊥

negat(val) =

let res = for bit in val then bit = ~bit

isequ(val1, val2) =

truth\_value = val1==val2? true: false

or(val1, val2) =

truth\_value = (val1 ==1 || val2 == 1)? true: false

and(val1, val2) =

truth\_value = (val1 ==1 && val2 == 1)? true: false

bgt(val1, val2) =

truth\_value = (val1 > val2) ? true: false

bge(val1, val2) =

truth\_value = (val1 >= val2) ? true: false

blt(val1, val2) =

truth\_value = (val1 < val2) ? true: false

ble(val1, val2) =

truth\_value = (val1 <= val2) ? true: false

**2.2.2 存储域**

Storable = Value

Store = Location → (stored Storable + undefined + unused)

存储域相关的辅助函数（在函数域中）：

empty\_store: Store

allocate: Store → Store × Location

deallocate: Store × Location → Store

update: Store × Location × Storable → Store

fetch: Store × Location → Storable

以上辅助函数形式化定义：

empty\_store = λloc.unused

allocate(sto) =

let loc = any\_unused\_location(sto) in

(sto[loc → undefined], loc)

deallocate(sto, loc) = sto[loc → unused]

update(sto, loc, stble) = sto[loc → stored stble]

fetch(sto, loc) =

let stored\_value(stored stble) = stble

stored\_value(undefined) = fail

stored\_value(unused) = fail

in

stored\_value(sto(loc))

**2.2.3 环境域**

Environ = Identifier → (bound Bindable + unbound)

Bindable = value Value + variable Location + class Class +

function Function + method Method + exception Exception

环境域相关的辅助函数（在函数域中）：

empty-environ: Environ

bind: ldentifier × Bindable → Environ

overlay: Environ × Environ → Environ

find: Environ × Identifier → Bindable

coerce: Store × Bindable → Value

以上辅助函数形式化定义：

enpty-environ = λI. unbound

bind(I, bdble) = λI'. if I'=I then bound bdble else unbound

overlay(env', env) =

λI. if env' (I)/=unbound then env' (I) else env (I)

find(env, I) =

let bound\_value (bound bdble) = bdble

bound\_value (unbound) = ⊥

in

bound\_value (env (I))

coerce(sto, find(env, I))

= val

| fetch(sto, loc)

**2.2.4 数组域**

Array\_Value = Value\*

Array\_Varibale = Variable\*

对数组变量操作相关的辅助函数（在函数域中）：

allocate\_array: Store × Natural → Store × Array\_Varibale

component: Integer × Array\_Varibale → Variable

fetch\_array: Store × Array\_Varibale → Array\_Value

update\_array: Store × Array\_Varibale × Array\_Value → Store

fetch\_variable: Store × Variable → Value

update\_variable: Store × Variable × Value → Store

以上辅助函数形式化定义：

component(int, nil) = ⊥ //取空序列只能失败

component(int, var, arrvar) = //其中用到的predecessor是为了递归调用时取出第int个arrvar元素的变量

if int = 0 then var

else component(predecessor(int), arrvar)

fetch\_array(sto, nil) = nil

fetch\_array(sto, var · arrvar) =

fetch\_variable(sto, var) · fetch\_array(sto, arrvar)

update\_array(sto, nil,nil) = sto

update\_array(sto, var·arrvar, val·arrval) =

let sto = update\_variable(sto, var, val) in

update\_variable(sto, arrvar, arrval)

fetch\_variable(sto, simple\_variable loc) = fetch(sto, loc)

fetch\_variable(sto, array\_variable, arrvar) =

array\_value(fetch\_array(sto, arrvar))

update\_variable(sto, simple\_variable loc, stble) =

update(sto, loc, stble)

update\_variable(sto, array\_variable arrvar, array\_value arrval) =

update\_array(sto, arrvar, arrval)

**2.2.5 类域**

Class\_Value = Value\* + Method\*

Class\_Variable = Variable\*

对类变量操作相关的辅助函数（在函数域中）：

get\_attr: Store × Class\_Variable × Variable → Value

set\_attr: Store × Class\_Variable × Variable → Store

get\_method: Store×Environ×Class\_Variable×Identifier → Store

以上辅助函数形式化定义：

get\_attr(sto, class\_variable clsvar, var) =

let sto1 = fetch(clsvar sto, loc) in

value(fetch(sto1, find(env, var)))

set\_attr(sto, class\_variable clsvar, var) =

let sto1 = fetch(clsvar sto, loc) in

update(clsvar env, sto1, find(env, var))

get\_method(sto, env, class\_variable clsvar, method\_name) =

let sto1 = fetch(clsvar sto, loc) in

find(clsvar env, method\_name)

**2.2.6 异常域**

异常域相关的辅助函数（在函数域中）：

catch: Environ × Store → Exception\_Value

以上辅助函数形式化定义：

catch(env, sto) = // get\_excption表示获取该语句块出现的异常

let excption\_value var = get\_excption(env, sto) in

let variable loc = allocate(sto') in

update(sto', loc, var)

**2.2.7 函数域**

值域、存储域、环境域、数组域、类域和异常域相关的辅助函数均写在了以上各自的章节2.2.1-2.2.6中，就不在此节中赘述。此节主要提及一些函数形参和实参相关的语义函数。辅助函数如下：

Function = Argument → Value

Function = Argument → Store → Value

bind\_parameter: Formal\_Parameter → (Argument → Environ)

give\_argument : Actual\_Parameter → (Environ → Argument)

以上辅助函数形式化定义：

形参的语义等式是：

bind\_parameter(value val) = bind (I, value val)

bind\_parameter(variable loc)= bind(I,variable loc)

实参的语义等式是:

give\_argument(E, env, sto) = value (evaluate E env sto)

give\_argument(I, env, sto) =

let variable loc = find (env，I) in

variable loc

**2.2.8 相关词法和语法的指称语义**

* 整数值和小数值

valuation[digit] = digit’ (digit∈Digit, digit’∈Natural)

valuation[numdigit]= 10 \* valuation[num] + digit’

valuation[num1.num2]= num1 + valuation[num2] / 10

* 数组声明

allocate\_variable[I[E\_LIST]] =

let allocate\_array(sto, size) =

let (sto', var) = allocate\_variable I sto in

if size==1 then (sto', var·nil)

else

let (sto'', arrvar) =

allocate\_array(sto', predecessor(size)) in

(sto'', var·arrvar)

in

let (sto', arrvar) = allocate\_array(sto, valuation length(E\_LIST)) in

(sto'', array\_variable arrvar)

* 函数声明

elaborate[I( [Parameter\_list] )B] env =

let func arg =

let parenv = bind\_parameter Parameter\_list arg in

evaluate B (overlay (parenv，env )) in

(bind(I, function func))

* 类声明

elaborate[class I B] env =

let cls =

let internal\_env = evaluate B (overlay(internal\_env, env)) in

(bind(I, class cls))

* 方法声明

elaborate[I( [Parameter\_list] ) B] env =

let method arg =

let parenv = bind\_parameter Parameter\_list arg in

evaluate B (overlay (parenv，env )) in

(bind(I, function func))

elaborate[I B] env =

let func arg =

evaluate B (overlay (env，env )) in

(bind(I, function func))

elaborate[I=(B1)B2] env =

let func arg =

let parenv = bind\_parameter evaluate(B1) arg in

evaluate B2 (overlay (parenv，env )) in

(bind(I, function func))

* 初等表达式

evaluate[IDENTIFIER] env sto = coerce(sto.find(env.I))

evaluate[false] env sto = turth\_value false

evaluate[true] env sto = turth\_value true

evaluate[null] env sto = value null

evaluate[int\_literal] env sto = integer int\_literal

evaluate[double\_literal] env sto = double double\_literal

evaluate[string\_literal] env sto = string string\_literal

* 基本表达式

evaluate[PE = LE] env sto =

let value val = evaluate LE env sto in

let variable loc = find(env, PE) in

update(sto, loc, val)

evaluate[PE += LE] env sto =

let value val1 = evaluate PE env sto in

let value val2 = evaluate LE env sto in

let value val3 = sum(val1 , val2)

let variable loc = find(env, PE) in

update(sto, loc, val3)

evaluate[PE -= LE] env sto =

let value val1 = evaluate PE env sto in

let value val2 = evaluate LE env sto in

let value val3 = difference(val1 , val2)

let variable loc = find(env, PE) in

update(sto, loc, val3)

evaluate[PE \*= LE] env sto =

let value val1 = evaluate PE env sto in

let value val2 = evaluate LE env sto in

let value val3 = mult(val1 , val2)

let variable loc = find(env, PE) in

update(sto, loc, val3)

evaluate[PE /= LE] env sto =

let value val1 = evaluate PE env sto in

let value val2 = evaluate LE env sto in

let value val3 = div(val1 , val2)

let variable loc = find(env, PE) in

update(sto, loc, val3)

evaluate[PE %= LE] env sto =

let value val1 = evaluate PE env sto in

let value val2 = evaluate LE env sto in

let value val3 = mod(val1 , val2)

let variable loc = find(env, PE) in

update(sto, loc, val3)

evaluate[LE1 || LE2] env sto =

let value val1 = evaluate LE1 env sto in

let value val2 = evaluate LE2 env sto in

ruth\_value(or(val1, val2))

evaluate[LE1 && LE2] env sto =

let value val1 = evaluate LE1 env sto in

let value val2 = evaluate LE2 env sto in

ruth\_value(and(val1, val2))

evaluate[RE1 == RE2] env sto =

let value v1 = evaluate RE1 env sto in

let value v2 = evaluate RE2 env sto in

truth\_value(isequ(v1, v2))

evaluate[RE1 != RE2] env sto =

let value v1 = evaluate RE1 env sto in

let value v2 = evaluate RE2 env sto in

~truth\_value(isequ(v1, v2))

evaluate[AE1 > AE2] env sto =

let value v1 = evaluate AE1 env sto in

let value v2 = evaluate AE2 env sto in

truth\_value(bgt(v1, v2))

evaluate[AE1 >= AE2] env sto =

let value v1 = evaluate AE1 env sto in

let value v2 = evaluate AE2 env sto in

truth\_value(bge(v1, v2))

evaluate[AE1 < AE2] env sto =

let value v1 = evaluate AE1 env sto in

let value v2 = evaluate AE2 env sto in

truth\_value(blt(v1, v2))

evaluate[AE1 <= AE2] env sto =

let value v1 = evaluate AE1 env sto in

let value v2 = evaluate AE2 env sto in

truth\_value(ble(v1, v2))

evaluate[ME1 + ME2] env sto =

let value v1 = evaluate ME1 env sto in

let value v2 = evaluate ME2 env sto in

value(sum(v1, v2))

evaluate[ME1 - ME2] env sto =

let value v1 = evaluate ME1 env sto in

let value v2 = evaluate ME2 env sto in

value(difference(v1, v2))

evaluate[UE1 \* UE2] env sto =

let value v1 = evaluate UE1 env sto in

let value v2 = evaluate UE2 env sto in

value(mult(v1, v2))

evaluate[UE1 / UE2] env sto =

let value v1 = evaluate UE1 env sto in

let value v2 = evaluate UE2 env sto in

value(div(v1, v2))

evaluate[UE1 % UE2] env sto =

let value v1 = evaluate UE1 env sto in

let value v2 = evaluate UE2 env sto in

value(mod(v1, v2))

evaluate[PE ++] env sto =

let value v1 = evaluate PE env sto in

value(sum(v1, 1))

evaluate[PE--] env sto =

let value v1 = evaluate PE env sto in

value(difference(v1, 1))

evaluate[-PE] env sto = value(mult(PE, -1))

evaluate[!PE] env sto = value(nagat(PE, -1))

* 表达式:赋值语句（变量第一次出现，未分配存储空间）

execute[I=E] env sto =

let val = evaluate E env sto in

let (sto’, loc) = allocate sto in

(bind(I,variable loc), sto’) in

(bind(I,value val), sto’)

* 表达式:数组取值

evaluate[PE[E]] env sto =

let variable (array\_variable arrvar) =

identify PE env sto in

let integer val =

evaluate E env sto in

component(val, arrvar)

* 表达式:类的属性调用

evaluate[PE.I] env sto =

let class\_variable val1 = evaluate PE env sto in

let variable var2 = evaluate I env sto in

value(get\_attr(sto, var1, var2))

* 表达式:函数调用

evaluate[PE( Argument\_list )] env sto =

let function func = find (env，PE) in

let arg = give\_argument Argument\_list env in

func arg

* 表达式:类的方法调用

evalute[PE.I( Argument\_list )] env sto =

let string clsname = String PE in

let function func = get\_method(sto, env, clsname, I)

let arg = give\_argument Argument\_list env in

func arg

* 条件语句

execute[if (E) B] env sto =

if evaluate E env sto = truth\_value true

then execute B env sto

execute[if (E) B1 else B2] env sto =

if evaluate E env sto = truth\_value true

then execute B1 env sto

else execute B2 env sto

execute[if (E1) B1 elsif (E2) B2] env sto =

if evaluate E1 env sto = truth\_value true

then execute B1 env sto

else if evaluate E2 env sto = truth\_value true

then execute B2 env sto

execute[if (E1) B1 elsif (E2) B2 else B3] env sto =

if evaluate E1 env sto = truth\_value true

then execute B1 env sto

else if evaluate E2 env sto = truth\_value true

then execute B2 env sto

else execute B3 env sto

* while循环

execute[while (E) B] env sto =

let execute\_while env sto =

if evaluate E env sto = truth\_value true

then execute\_while env (execute B env sto)

else sto

in

execute\_while

* for循环

execute[for (E1; E2; E3) B] env sto =

execute E1 env sto

let execute\_for env sto =

if evaluate E2 env sto = truth\_value true

then execute\_for env (execute B env sto

execute E3 env sto)

else sto

in

execute\_for

* for each循环

execute[foreach (I:E) B] env sto =

let value val = evaluate E env sto in

let variable loc = find(env, I) in

update(sto, loc, val)

let value point = 1

let execute\_foreach env sto =

if evaluate I!=null = truth\_value true

then execute\_foreach env (

execute B env sto

let value val = evaluate E[point] env sto in

let variable loc = find(env, I) in

update(sto, loc, val)

point = sum(point, 1)

)

else sto

in

execute\_foreach

* break语句

execute[break;] env sto =

break

* continue语句

execute[continue;] env sto =

continue

* return语句

execute[return E;] env sto =

let value var = evaluate E env sto in

let variable loc = allocate(sto') in

update(sto', loc, var)

* 异常处理

execute[try B1 catch (I) B2 finally B3] env sto =

let exception\_value i = catch(execute B1 env sto) in

if i in Exception\_Value

then execute B2 env sto

execute B3 env sto

execute[try B1 finally B2] env sto =

let exception\_value i = catch(execute B1 env sto) in

execute B2 env sto

execute[try B1 catch (I) B2] env sto =

let exception\_value i = catch(execute B1 env sto) in

if i in Exception\_Value

then execute B2 env sto

execute[throw E ;] env sto =

let exception\_value i = catch(env, sto) in

let variable loc = find(env', E) in

update(sto, loc, i)

1. 语言的代码范例

【内容概要】需根据设计的语言的语法和语义来给出语言的代码示例。

* 代码1

class EM{

q=5;

create\_point(x, y) {

return this;

}

static test\_static\_method(){

}

}

print("hhh");

* 代码2

############################################################

#test fuction defination

import nunmpy;

global a,b,c;

function show(){

for(i=0;i<3;i=i+1){

print(i);

}

}

function show2(){

for(i=0;i<4;i=i+1){

print(i);

}

}

#test funtion call

show();

show2();

import animal;

#test class defination

class Dog{

name;

#static str="wolf";

#test method def

new(name1){

name=name1;

}

call(){

print(str);

}

eat(food){

}

}

#test class method call

dog1 = Dog.new("dog1");

dog1.eat("meat");

#test calculation

a=(2+5)\*2/2-1;

a++;

a--;

b={1,2,3};

c={a,b};

p = create\_point(10, 20);

#test try catch

try {

a=1/0;

} catch (ex) {

desc\_exception(ex);

}

if(a==6){

print("== successful");

}else{

print("== error");

}

if(a<=6){

print("<= successful");

}else{

print("<= error");

}

if(a>=6){

print(">= successful");

}else{

print(">= error");

}

# test elsif

if(a>=6){

print(">= successful");

}elsif(a==4){

print("a == 4");

}elsif(a==3){

print("a == 3");

}

else{

print(">= error");

}

#test for while loop

for(i=0;i<10;i++){

while(i<5){

print(i);

}

if(i == 1){

continue;

}

elsif(i == 9){

break;

}

}

* 代码3

class Person{

name, age, sex, ID, address;

new( \_name, \_age, \_sex, \_ID){

name = \_name;

age = \_age;

sex = \_sex;

ID = \_ID;

}

setAddress(\_address){

address = \_address;

}

getUp(){

print(name);

print(": get up.\n");

}

haveBreakfast(food){

print(name);

print(": have breakfast of ");

print(food);

print(".\n");

}

takeRest(){

print(name);

print(": take a rest.\n");

}

celebrateBirthday(){

age += 1;

print(name);

print(": celebrate a birthday.\n");

return age;

}

compareAge(p){

if(this.age==p.age){

print("The same age.");

}

elsif(this.age>p.age){

print(this.name);

print(" is older than ");

print(p.name);

}

else{

print(this.name);

print(" is younger than ");

print(p.name);

}

}

}

function getLength(arr){

size = 0;

foreach(num:arr){

size = size +1;

}

return size;

}

function getSum(arr){

res = 0;

foreach(num:arr){

res += num;

}

return res;

}

function getMean(arr){

return getSum(arr) / getLength(arr);

}

function findPerson(name, arr){

i = 0;

flag = 0;

for( ;i < getLength(arr); i++){

if(arr[i].name == name){

flag = 1;

return arr[i];

}

}

if(flag == 0){

print("Not Found!");

}

}

p1 = Person.new("Lee", 22, "male", "123456");

p2 = Person.new("Rey", 20, "female", "654321");

p2.setAddress("267 N 4th Ring Middle Rd, Beijing");

p3 = Person.new("Olaf", 5, "male", "111111");

p4 = Person.new("Elsa", 24, "female", "222222");

p5 = Person.new("Anna", 23, "female", "333333");

people = {p1, p2, p3, p4, p5};

people[0].getUp();

people[0].haveBreakfast("egg and milk");

people[1].celebrateBirthday();

people[3].compareAge(people[4]);

* 代码4

max\_index = 9;

class a{

attr\_a;

new(\_attr\_a){

attr\_a = \_attr\_a;

}

aprint(){

print("aaaa");

}

}

class b extend a{

attr\_b,attr\_2;

new(\_attr\_b, \_attr\_2){

attr\_b = \_attr\_b;

}

bprint(){

print("bbbb");

}

}

class tEst\_I{

de1f;

new(\_de1f){

de1f = \_de1f;

}

}

class TeSt\_INde extend tEst\_I{

attr\_a,attr\_2;

new(\_attr\_a, \_attr\_2){

attr\_a = \_attr\_b;

}

a(){

print("aaaa");

}

}

print("hhh");

intb = {0,1,2,3,4,5,6,7,8,9};

try{

for( i = 0; i < intb[max\_index]; i++){

if(i %2 == 0){

print(intb[i]);

}

else{

print("mod!=0");}

}

}

catch(INT\_ERROR){

print("error");

}

finally{

print("end");

}

i = 0;

while(i < intb[max\_index]){

A = 1;

B = 2;

if(!A || B <= 2){ A += 1;}

elsif(A != 2){ A -= 1;}

elsif(B == 0){ B \*= 3;}

elsif(B == 0){ B /= 3;}

elsif(B == 0 && A ==1){ B %= 10;}

}

* 代码5

#test function defination

function isPrime(num){

tmp=num-1;

for(i=2;i<tmp;i=i+1){

if(num%i==0){

print("no");

return false;

}

}

print("yes");

return true;

}

#test function call

for(j=1;j<100;j=j+1){

if(isPrime(j)){

print(j+"is Prime");

}

}

#test class def

class Person{

name,age,address;

}

class Student extend Person{

sid,

classroom = "101",

score,

rank;

new(name1,sid1,score1){

name=name1;

sid=sid1;

score=score1;

}

show\_score(){

print(name+"'s score is"+score);

}

}

#test return

function max(num1,num2){

if(num1>num2){

return num1;

}

else{

return num2;

}

}

#test class call

stu1=Student.new("1","Kiki",99);

stu2=Student.new("2","Chuchu",74);

stu3=Student.new("3","Lily",93);

stu4=Student.new("4","Dode",66);

student[5]={stu1,stu2,stu3,stu4};

maxScore=stu1.score;

topId=1;

#test for

for(k=1;k<4;k++){

if(student[k]>max){

maxScore=student[k];

topId=k;

}

}

print(topId+"get the max score:"+maxScore);

#test while

t=0;

while(t<4){

if(student[t].score>=98){

continue;

}

student[t].score+=5;

}

#test foreach

foreach(m:student){

if(student[m].score>90){

student[m].rank='A';

}

elsif(student[m]>75){

student[m].rank='B';

}

else{

student[m].rank='C';

}

}

* 代码6（一个错误代码样例）

print("I am Sython");

print("who are you")

print("I am Sython2");

以下代码解释器可执行

* 代码7

############################################################

function show(){

for(i=0;i<3;i=i+1){

print(i);

}

}

function show2(){

for(i=0;i<4;i=i+1){

print(i);

}

}

show();

show2();

* 代码8

function fib(){

n1=1;

n2=2;

for( i=3;i<=10;i=i+1){

n3=n1+n2;

print(n3);

n1=n2;

n2=n3;

}

}

fib();