CS61A

——一点学习笔记

Jabofish

2024 年 2 月 19 日

Preface

During the winter vacation of my freshman year, I participated in the study of this course and planned to make some records through this document.

It is based on resourses from https://inst.eecs.berkeley.edu/~cs61a/fa20/

repository:https://github.com/Jabofish/CS61A/

Jabofish

前言

在大一寒假期间，我开始学习这门课程，并用这篇文档来做记录。

资料来源于 https://inst.eecs.berkeley.edu/~cs61a/fa20/

仓库：https://github.com/Jabofish/CS61A/

Jabofish

2024 年 2 月 19 日

目录

1 Week-1

1.1 Computer Science

Computer Science is the study of what problems can be solved using computation, how to solve those problems, and what techniques lead to effective solutions.

1.1.1 Lab 00

课程作业的提交涉及到 Git 的安装与使用，需要给电脑安装 Git，并掌握基本的 git 命令。

使用方法：

1. 打开 Git Bash
2. 使用命令行语句进入指定目录
3. 使用 python 来打开文件，自动完成代码检测

涉及的命令行语句：

```
ls 列出当前目录中的所有文件
cd <path to directory> 切换到指定的目录中(cd .. 表示回退)
mkdir <directory name> 用给定的名字建立一个新的目录
mv <source path> <destination path> 将指定来源的文件移动到指定的目的地
python ok -q python-basics -u --local 运行测试
python ok --local 代码测试
```

1.2 Functions

1.2.1 Lecture

表达式的分析顺序：先看操作符（或对应的函数），再看操作数（也可以是一个基元表达式）。赋值语句的执行顺序：1. 等号右侧从左到右计算。2. 将右侧的结果赋值到左侧。

```
a = 1
b = 2
b,a = a+b,b
print(a,b)
```

以上代码将会输出 2 3

赋值和函数定义都是一种抽象方式，后者更为强大。找声明时优先在本地框架找，找不到再去全局框架找。

1.2.2 Hw 01

需了解 operator 库里的常见函数：加法 a + b add(a, b) 减法 a - b sub(a, b)，并初步理解函数的抽象性；了解 min 和 max 的运用

关于 git 的上传的重要补充：

基本流程：工作区-add-暂存区-commit-本地仓库区-push-远程仓库区

```
#克隆
git clone http://xxx
```

```
#拉取
git pull http://xxx

#添加
git add xxx

#描述信息
git commit -m "注释"

#推送到远程
git push origin master

git push
```

如果开了代理还是提交失败，可能是 git 未使用代理，需设置代理：

git config –global http.proxy 127.0.0.1:7890（从系统设置查看 IP 和端口）

在 vsc 中提供了 git 的可视化操作，极大降低了对命令行的要求，可以考虑使用。

2 Week-2

2.1 Control

2.1.1 Lecture

Python 中每个函数都有返回值，当没有明确返回值时将会返回默认的 None（None 表示空，是隐性的）。纯函数只有返回值，非纯函数会产生别的效果（如打印值，即 display value）。

```
print(print(1))
result is:
1
None
```

一个程序会有多个框架，包括全局框架和局部框架（如函数体内部），相同的变量名在不同的框架内可能拥有不同的含义。可以根据子框架追溯到父框架，变量和函数优先在子框架中找到定义，找不到再去父框架找。

运算符可以看作是运算函数的简记符。truediv:/ floor:// mod:%

一个函数可以有多个返回值，用逗号隔开的返回值会按顺序返回。doctest 可以通过文档内的提示语句来实现对代码的运行过程和结果进行测试，检验是否符合预期。

doctest 的调用：python -m doctest -v example.py

doctest 的写法：在函数的注释中写入：> > > 函数（值），回车写入预期结果。条件语句的基本写法：

```
if exp:
    clause
elif exp:
    clause
else:
    claude
```

while 语句：

```
    i, total= 0, 0
    while i < 3:
        i = i + 1
        total = total + i
```

例子，质因数分解：

```
    def prime_factors(n):
    while n > 1:
        k = smallest_prime_factor(n)
        n = n // k
        print(k)
def smallest_prime_factor(n):
    k = 2
    while n % k != 0:
        k =k + 1
    return k
prime_factors(858) # case
```

这个例子的绝妙之处在于，通过循环实现了类似于递归的效果。可以先把一个大功能划分成若干个小功能，再逐一实现。

2.1.2 Lab 01

要学会使用把数字转成字符串，然后用 for 遍历每个数字的办法。

2.2 Higher-order Functions

以斐波那契数列为例：

```
    def fib(n):
 """Compute the nth Fibonacci number, for N >= 1."""
 pred, curr = 0, 1 # 0th and 1st Fibonacci numbers
 k = 1 # curr is the kth Fibonacci number
 while k < n:
 pred, curr = curr, pred + curr
 k = k + 1
 return curr
```

assert 语句的运用：

```
    assert exp,'Hint'
    #Do nothing if exp is True
    #Return an ERROR and 'Hint' if exp is False
```

对函数划分的精妙运用：

```
    def area(r, shape_constant):
    """Return the area of a shape from length measurement R."""
    assert r > 0, 'A length must be positive'
    return r * r * shape_constant

def area_square(r):
    return area(r, 1)
```

```
def area_circle(r):
    return area(r, pi)

def area_hexagon(r):
    return area(r, 3 * sqrt(3) / 2)

# Functions as arguments
```

从上面的例子可以学到：把相同点抽象出来作为一个函数框架，然后在具体函数中调用并补全这个框架。在函数框架中使用另一个函数名或一个常数作为参数，可以实现不同的运算法则。

```
    def make_adder(n):
        def adder(k):
            return k + n
        return adder
'''
在上例中，n作为形式参数在adder的定义中起了作用。
当n=2时，返回的adder函数中所定义的return语句是：return k + 2。
所以make_adder(1)(2)的值为3，
此时make_adder(1)(2)等效于adder(2)，
且这里的adder返回语句为：return k + 1。
'''
```

函数可以作为参数，把函数入口作为返回值，可以在全局框架中被调用。这就是高阶函数的应用。

lambda 表达式：（匿名函数的使用）

1.square = lambda x: x * x（参数: 表达式）

2.(lambda x: x * x)(4)（可直接调用）

3.def compose1(f, g):

return lambda x: f(g(x))

```
    def search(f):
        x = 0
        while not f(x):
            x += 1
#一个简单的通过返回值来决定结束的遍历函数（学会对返回值的运用）
    def inverse(f):
        return lambda y: search(lambda x: f(x) == y)
#用非常简单的代码实现了整数范围内的反函数的值的寻找
    sqrt = inverse(square)
```

自带函数的控制语句具有调用表达式无法实现的作用，因为调用表达式总是先计算表达式的值。

在逻辑判断中会有“短路”现象（这也是一种控制），以避免不必要的运算，类似于 c 语言。因此需掌握 and，or，not 的用法。

A and B: 两者都正确，返回 and 后面的值; 两者有一个错误，返回最先错误的。

A or B: 返回其中布尔值为 True 的一方; 若无，返回 or 后面的值。

也就是说：在哪里停下就返回哪里的结果。

```
    <consequent> if <predicate> else <alternative>
    #条件表达式，如果if后为真，执行前面的语句，否则执行else后的语句，例子如下
    1/x if x!=0 else 0
```

2.2.1 Hog

这是一个很有意思的项目，做成后很有成就感，不过中间遇到了很多的问题。

1. 在 problem6 中写错了函数的位置，但是意外通过了测试点，导致后面在写 problem7 的时候始终没找到 bug，要十分小心。(后来发现其实是测试文件还没解锁，自己没仔细看以为是通过了，难绷)

2.problem7 中以高阶函数的形式修改了原函数参数的默认值，非常巧妙。

3. 高阶函数使用时给参数赋值的顺序也很有讲究，会影响逻辑结构。

2.3 Environments

高阶函数的特点：以函数作为参数，或返回函数，后者用到嵌套定义。（子框架中用到的未定义参数会到父框架中找值）

在已有一个函数的情况下，仅用该函数作为参数来定义函数，使用多层嵌套来返回被定义的函数时可以实现一个效果：新定义的函数可以通过多次赋值来逐层解开嵌套。如定义一个以 f0 为参数的 f1，然后在 f1 中定义 f2，f2 中定义 f3，并且 f3 返回占位赋值后的 f0 的结果（事实上还未赋值，只是用参数先占住了位置），f2 返回 f3，f1 返回 f2，并用 fun1 来存储最后这个返回值。先调用 fun，将 f0 代入，接下来，调用 fun1，代入的参数会补全 f2 的空缺部分，然后返回 f3（此时的 f3 和前面的 f3 已有不同），并用 fun 储存这个返回值。继续调用 fun，代入的参数将同前面的参数一起补全 f0 所需的参数，完成 f0 的完全调用，并返回 f0 的结果。

这是 currying 的应用。柯里化（Currying）是一种处理多元函数的方法。它产生一系列连锁函数，其中每个函数固定部分参数，并返回一个新函数，用于传回其它剩余参数的功能。

```
fun = lambda f0: lambda x: lambda y: f0(x,y)
#或者
def curry2(f):
    """Return a curried version of the given two-argument function."""
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

2.3.1 Lab 02

简单的高阶函数运用。

3 Week-3

3.1 Design

函数命名最好能反映其效果、行为、返回值，见名知义。给重复使用的表达式或过程进行命名。

3.2 Function Examples

一些实例，用于应付考试。通过代入特定的值来理顺程序的执行内容/

4 Week-4

4.1 Recursion

递归函数：直接或间接地调用自己的函数。通常从判断基本情况开始，基本情况的判断是不需要递归的；更复杂的情况则会通过调用自己来一步步地化简到基本情况。基本情况可以不止一种。递归可以理解为数学归纳法的实际应用。因此可以用数学归纳法来设计和检验递归函数。当有奇偶数区分时可以设计两个递归函数分别调用对方。递归和迭代（循环）往往可以相互转化，但递归往往更简单。

4.2 Tree Recursion

```
    def factorial(n):
        if n == 0:
            return 1
        else:
            return n * factorial(n-1)
```

```
    def cascade(n):
        print(n)
        if n > 10:
            cascade(n//10)
            print(n)
```

树递归：递归函数体中有多次对自身的调用。如斐波那契数列：

```
    def fib(n):
        if n == 0:
            return 0
        else if n == 1:
            return 1
        else:
            return fib(n-2) + fib(n-1)
```

步骤较多的递归可以通过记值来简化计算过程。

使用递归可以写出汉诺塔每次移动的步骤。

```
    def print_move(origin, destination):
    """Print instructions to move a disk."""
    print("Move the top disk from rod", origin, "to rod", destination)

def move_stack(n, start, end):
    """Print the moves required to move n disks on the start pole to the end
    pole without violating the rules of Towers of Hanoi.

    n -- number of disks
    start -- a pole position, either 1, 2, or 3
    end -- a pole position, either 1, 2, or 3

    There are exactly three poles, and start and end must be different. Assume
    that the start pole has at least n disks of increasing size, and the end
    pole is either empty or has a top disk larger than the top n start disks.
    """
```

```
    assert 1 <= start <= 3 and 1 <= end <= 3 and start != end, "Bad start/end"
    if n == 1:
        print_move(start, end)
    else:
        spare_peg = 6 - start - end
        move_stack(n-1, start, spare_peg)
        print_move(start, end)
        move_stack(n-1, spare_peg, end)
```

4.2.1 HW 02

递归函数也可以越递归参数越大，只需要写一个高阶函数再调用，就可以实现和 while 语句相近的效果。

在没有函数名的情况下也可以通过 lambda 语句实施递归操作，但是需要将函数自身作为参数传递到每一次的调用之中。

```
    from operator import sub, mul
    def make_anonymous_factorial():
    """Return the value of an expression that computes factorial.

    >>> make_anonymous_factorial()(5)
    120
    >>> from construct_check import check
    >>> # ban any assignments or recursion
    >>> check(HW_SOURCE_FILE, 'make_anonymous_factorial', ['Assign', 'AugAssign', 'FunctionDef', 'Recursion
                                                           '])
    True
    """
    return (lambda f:lambda n:f(f,n))(lambda f,n: 1 if n == 1 else mul(n, f(f,sub(n, 1))))
```

5 Week-5

5.1 Containers

lists: 列表，[]: 列表之间可以直接相加和相乘。可以用 in 和 not in 判断某个元素是否在列表中。用'for element in list：' 来遍历列表。（element 可以是多个参数，用于读取嵌套列表中的元素）

range(start,end) 前闭后开，用 list(range(start,end)) 可以构造列表，只含区间内的整数。[expression for element in list] 可以对已有列表的基础上通过表达式写出新的列表，也可以取出列表里特定序号的元素。[expression for element in list if condition] 可以按条件筛选元素。

strings: 字符串，' ' " "。

用 exec 可以执行字符串中的表达式，用序号取字符串的元素得到的还是字符串。用 in 表达式可以判断某个单词或某个部分是否在字符串中出现。

5.1.1 Lab 04

对整数进行逐位递归处理时可以先从第一个开始，也可以先从最后一个开始。

5.2 Data Abstraction

分数可以看成是由分子和分母构成的复合数据，计算时可以分别得到分子和分母，以保证精确度，如分数的加减法。

要保证代码块的抽象化，需要使用函数来封装功能，避免出现难以理解的数字和实质。

```
    def rational(n, d):
    """A representation of the rational number N/D."""
    g = gcd(n, d)
    n, d = n//g, d//g
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select

def numer(x):
    """Return the numerator of rational number X in lowest terms and having
    the sign of X."""
    return x('n')

def denom(x):
    """Return the denominator of rational number X in lowest terms and positive."""
    return x('d')
```

在不使用列表的情况下实现对分子分母的返回（涉及高阶函数）。

dictionaries:key:value，字典，无序。用 keys/values/items 来取出所有的元素（键/值/对）。dict.get(key,0) 可以返回对应的键，找不到则默认返回后面的 0。[key:expression for element in range] 可以快速构建字典。

5.2.1 Cats

写一个打字练习器。里面有一个关于递归的处理很巧妙：

```
        add_diff = 1+pawssible_patches(start,goal[1:],limit-1)
        remove_diff = 1+pawssible_patches(start[1:],goal,limit-1)
        substitute_diff = 1+pawssible_patches(start[1:],goal[1:],limit-1)
        return min(add_diff,remove_diff,substitute_diff)
```

形成像树一样的分支结构。

5.3 Trees

框-指针表示法：可用于表示树结构。嵌套列表是一种典型的树结构。

切片：[start:end:step] 前闭后开。

序列内置了 sum/max/all 函数，可以直接使用。

A tree has a root label and a list of branches.Each branch is a tree.

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return [label] + list(branches)
```

```
#树的拼接
def label(tree):
    return tree[0]
#取标签
def branches(tree):
    return tree[1:]
#取分支
def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
#判断是否是树
def is_leaf(tree):
    return not branches(tree)
#判断是否是叶
```

6 Week-6

6.1 Binary Numbers

二进制数在计算机中存储比其他进制的稳定。负数以补码形式参与计算。补码的补码是原码。100 的补码还是 100，从而避免了-0 的出现。

布尔逻辑：and/or/not。通过串联和并联分别构建 and 和 or。另开一条带电阻的通路可以构建 not。

6.1.1 Lab 05

```
    return tree(label(t),[tree(leaf) for leaf in leaves])
    #这个可以一次性给树（原来是叶子）加多片叶子
    return [[num,fn(num)] for num in seq if fn(num)>=lower and fn(num)<=upper]
    #筛选
```

6.2 Circuits

设计电路：1. 列真值表。2. 为每个输出列的每一种情况写逻辑表达式（这个逻辑表达式必须保证只有一种输入才能得到该情况下的输出，即形成充分必要条件，只看输出为 1 的情况即可）。3. 组合每列所有情况来得到适用整个列的表达式。4，画电路图。

通过层层抽象来在简单结构的基础上构建更复杂的结构，但是简单抽象后的电路未必是资源优化最好的。

加法器：需要处理加法与进位，三进二出。

卡诺图化简法（Karnaugh maps）：用二维的方格图来简化逻辑（电路），尽量保证图表的坐标轴上相邻两个输入有一位及以上的相同位，可以更好地找到更有代表性地逻辑表达式。

6.2.1 HW 03

树与二叉树的转换，数据体的抽象化。

6.3 Mutable Values

对象：含有绑定在变量和函数的属性（数据和行为）。对象储存在类中。字符串的常用方法：upper/lower/swapcase

ord()：输出 ASCII 码

列表方法：append（追加一个元素）/extend（另一个列表的元素进入列表）/pop（移除并返回最后一个元素）/remove。加和或切分会在已有元素的基础上产生新列表，里面的嵌套列表依旧指向原对象。list()：也在已有基础上产生新列表，但是属于一个新的对象。

元组（tuples）：()，不可更改，可以作为字典的键。可以更改元组中的可变容器，如列表。

is：可以用于判断两个变量是否指向同一个对象。

```
t=[1,2,3]
t[1:3]=[t]
t.extend(t)
#t=[1,[...],1,[...]]
t=[[1,2],[3,4]]
t[0].append(t[1:2])
#t=[[1,2,[[3,4]]],[3,4]]
```