

问题求解课笔记

AUGPath

2023 年 2 月 21 日

目录

第一章 Python 基础回顾	5
1.1 问题的提出: 通过程序, 可以求解的问题	5
1.1.1 为什么学习计算机编程	5
1.1.2 程序设计的语法与语义	5
1.1.3 Python 代码执行可视化: 一个网站	6
1.1.4 可以执行任何程序的最小指令集	6
1.1.5 Python 中的整个世界	8
1.1.6 函数: 整合相似过程	12
1.2 一些小例子	13
第二章 命题的逻辑	19
2.1 问题的提出: 为什么要研究命题的逻辑	19

第一章 Python 基础回顾

1.1 问题的提出：通过程序，可以求解的问题

上学期我们学习了 Python 程序设计，可能我们会对学程序设计合理性产生质疑。

1.1.1 为什么学习计算机编程

看上去，计算机是一个笨重的工具，在时间或空间限制下可以有效地执行许多重复的任务。到现在为止，计算机还不能分析问题并提出解决方案。另一方面，人类在分析和解决问题方面非常出色，但很容易对重复的任务感到厌倦。

人类通过利用他们的分析和解决问题的技能，可以为可计算的问题提出算法（有限的指令，与有限的输入一起工作，产生输出）。然后，计算机可以遵循这些指令并产生一个答案。

你可以将你生活中的各种多余的任务自动化。一点点的 Python 可以给你的生活带来神奇的效果。甚至提高你的生产力。

1.1.2 程序设计的语法与语义

我们在学习英语的时候很重要的一部分是语法：也就是什么样的语言是可以被接受 (acceptable) 的。比如下面这个英文句子是没有语法错误的：

`Fuhai Zhu said that this test is only a small test, so don't panic.`

但是这句话不同人有不同的理解方式，这就是这句话的语义。

- 我们可以推断朱富海老师在安抚准备参加小测试学生们的情绪
- 但是南京大学数学系的同学知道这是一个有名的梗：上学期教高等代数的朱富海老师把线上期中考试叫做“小测验”，并在同学询问考试范围的时候微笑的答道：“从小学学的都考。”

简而言之（不严谨），现在我们有一个由一堆字符串和推导规则组成的形式系统，语法决定了这个形式系统能生存什么样的字符串，而至于这些字符串有什么样的含义则是语义的范畴。语法类似材料，语义类似与材料组成的各种建筑物，我们可以通过语法研究语义层面的推导，同时也可以从语义层面捕获语法中内涵的结构，其实语法和语义是相互区别又紧密联系，即从范畴论的角度看语法和语义是伴随的（其实不同的人做数学证明可以有不同的风格：偏语法和偏语义，不过大部分数学家更喜欢语义风格的证明，可能因为更直观，更容易被人脑接受~~~）

作者: 知乎用户链接: <https://www.zhihu.com/question/31347357/answer/892133941>

来源: 知乎著作权归作者所有. 商业转载请联系作者获得授权, 非商业转载请注明出处.

1.1.3 Python 代码执行可视化: 一个网站

Tools 工具

这是Python 代码执行可视化的机器, 我们可以用一个小程序来测试之.

```
1 for i in range(10):
2     print(i**2)
```

点击 Visualize Execution 就可以了, 你可以点击 Next 来继续模拟执行下一步.

在这里, 你可能会看到很多新奇的名词: 什么是 Global Frame, Object? 暂时先不用管. 不过你确实可以看到点击 Next 的时候 Print output 一栏一步一步的模拟了你的代码.

可以知道, 代码按照行数执行, 一次执行一行, 每一次执行计算机内部结构的状态 (右侧的面板). 下面我们化繁为简, 来看一看一个系统 (数学意义上) 能够完成任何人类完成的操作需要的最小可能的操作是什么.

1.1.4 可以执行任何程序的最小指令集

像数学的公理体系那样, 我们自然希望得到一个最小的指令集合, 并且我们可以用来写出任何的程序. 我们不妨从日常生活中找一点灵感吧.

Example 例子:

(等红绿灯) 观察红绿灯, **如果是**绿灯, 那就通过这个路口; **否则**继续等待. (遵纪守法的好公民)

(做作业) 明确今天的作业范围, 从**第一题**开始写, 写完题目**或者**一题目没有思路之后做**下一道题**, **直到**做完所有的问题.

(排序成绩单) 获得班上同学的所有**成绩单**, 拿一张新的白纸打好**表格**, **每一次**从成绩单中选取最大的分数, 把那一行**抄写**到新的白纸上. 之后把原来那张纸上的内容划去. **一直重复下去**, **直到**原来的成绩单上没有任何可以被划去的内容.

我们需要找一些东西来具象化我们脑子中的“红绿灯的状态”, “现在在做作业的题目编号”, 这些内容, 因此我们就希望把这些抽象出来. 因此我们有了变量的概念, 也就是值存在的空间.

把上面的三个内容转化成伪代码 (不唯一) 就是:

```
----- GO THROUGH CONJUNCTION -----
if traffic light's color is green:
    go pass by
else
    wait

----- DO HOMEWORK -----
```

```

range = [a..b]
working on problem = a
while working on problem <= b:
    finished this problem or can't work out
    working on problem = working on problem+1

----- SORT EXAM SCORE -----
list = get the source table
result = empty(for now)
while list is not empty
    k= get the max element of list
    write k to the next line of result
show result

```

其实要是能够构造出任何程序的原材料并不复杂. 无非就是变量的赋值, 判断, 跳转, 终止. 也就是, 如果你能声称有一套系统可以自动化的解决这四个内容, 那么这个系统就具有机械化地做任何人类做的事情. 换句话说, 你可以用这个工具创造整个世界.

其实最早这个想法是在计算机诞生之前人们孜孜以求的问题. Alan Turing 在 1936 年就提出了这样的设想. 他就是由只一条 (无限长) 的纸带和一根笔 (可以改纸带的内容, 并且查看纸带的内容并据此做判断), 并且有一个程序 (墙上的表格), 指示下一步要往哪转移. 只要能够移动读写头, 写纸带的某一个格子, 读纸带的某一个格子, 跳转, 以及终止, 这个机器就和我们人类的计算能力等价.

这是最早的图灵机的原型.

Example 例子:

运行上面图片的程序, 左右按照我们的左右进行 (规定 ABC 右移一格是 ABC).

(1) 现在机器的状态是 A(头部的字母), 看到的是 1(放大镜的字母)

(2) 于是把当前的格子改为 1, 纸带向右移动一格, 然后停机.

假设当前纸带的放大镜看到的是 0, 再运行一次:

状态:A 纸带状态: 0 1 1 1 0 1 1 0 0

(1) 现在机器的状态是 A(头部的字母), 看到的是 0(放大镜的字母), 执行第一行第一列的指令 1(改为 1) → (向右移动一格)B(状态改为 B) .

状态:B 纸带状态: 0 1 1 1 1 1 1 0 0

(2) 现在机器的状态是 B(头部的字母), 看到的是 1(放大镜的字母), 执行第二行第二列的指令 1(改为 1) → (向右移动一格)B(状态改为 B) .

状态:B 纸带状态: 0 1 1 1 1 1 1 0 0

(3) 现在机器的状态是 B(头部的字母), 看到的是 1(放大镜的字母), 执行第二行第二列的指令 1(改为 1) → (向右移动一格)B(状态改为 B) .

状态:B 纸带状态: 0 1 1 1 1 1 1 0 0

(4) 现在机器的状态是 B(头部的字母), 看到的是 0(放大镜的字母), 执行第二行第一列

的指令 0(改为 0) \rightarrow (向右移动一格)C(状态改为 C) .

状态:C 纸带状态: 0 1 1 1 1 1 1 0 0

(5) 现在机器的状态是 C(头部的字母), 看到的是 0(放大镜的字母), 执行第三行第一列的指令 1(改为 1) \leftarrow (向左移动一格)C(状态改为 C) .

状态:C 纸带状态: 0 1 1 1 1 1 1 0 1

(6) 现在机器的状态是 C(头部的字母), 看到的是 0(放大镜的字母), 执行第三行第一列的指令 1(改为 1) \leftarrow (向左移动一格)C(状态改为 C) .

状态:C 纸带状态: 0 1 1 1 1 1 1 1 1

(7) 现在机器的状态是 C(头部的字母), 看到的是 1(放大镜的字母), 执行第三行第二列的指令 1(改为 1) \leftarrow (向左移动一格)A(状态改为 A) .

状态:A 纸带状态: 0 1 1 1 1 1 1 1 1

(8) 现在机器的状态是 A(头部的字母), 看到的是 1(放大镜的字母), 执行第一行第二列的指令 1(改为 1) \rightarrow (向右边移动一格) \uparrow (停机)

状态:A 纸带状态: 0 1 1 1 1 1 1 1 1

下面我们来看一看为什么说可以用 Python 创造整个世界. 在这之前, 我们先来了解一下如何获取帮助和得到通俗易懂的教程.

请认真阅读并实践 (无论是在脑子还是在交互器里面) 文档 <https://docs.python.org/zh-cn/3/tutorial/introduction.html> 的内容. 可以让你了解更多易于理解的东西. 如果文章中有描述 C 和 Pascal 的句子, 忽略它就可以.

Idea 启示

摒弃“我一定要一次把所有的东西都弄懂”的理念, 因为这些知识十分的巨大. 所以先了解一部分作为正确的立足点, 然后慢慢扩大就行了. 这需要很长时间的积累, 千万不要急. (着急也没用, 只会加大精神内耗.)

1.1.5 Python 中的整个世界

下面的内容其实不用单独记忆, 只要明确有哪些语句, 这些语句造成的效果是什么就行了. 如果看到有任何的问题, 可以去搜一搜词典.

下面会有两个术语 (term), 分别是表达式 (expression) 和过程 (procedure), 表达式可以暂且认为是形如 $x+12$, $[2]*3$ 这样的可以进行计算的内容, 过程就是一系列执行的过程, 不一定要能得到值. 我们用一个例子感受一下.

```
def InsertionSort(A):
    for j in range(1, len(A)):          #Proc
        key = A[j]                      #A[j],key are expr #Proc #|
        i = j - 1                      #j-1,i are an expr #v  #|
        while (i >=0) and (A[i] > key):  #|
            #    <-expr->    <----expr----->          #|
            #    <-----expr----->                  #|
            A[i+1] = A[i] #A[i]is expr    #Proc    #|
```



```

        i = i - 1      #i-1 is expr    #v      #|
    A[i+1] =      key                                #v
    <-expr->    <-expr->

```

其实上面的 Proc 表示过程, 然后右边的是一个字符画, 表示 \downarrow , <-expr-> 其实表示的意思是 example 这一段是表达式.

expr

重要的是, 把示例代码放到上面提到的可视化网站里面看一看就会很清楚, 很多概念都是不用记忆的.

变量的定义与赋值

定义 1.1.1. (变量的赋值) 变量名 = 变量的值

语义:

下面我们给出注解:

- 在不加修饰的情况下, 变量的名称只在当前的缩进块内有效
- 命名是用来指代对象的. 这就是为什么有时候可视化工具里面 Frames 后面有一个箭头指着 Objects.
- 如果用一个变量 = 另一个变量, 大多数情况是现计算出来右手边表达式的值之后给左手边的变量. 有时候一些文章里面写作 $lhs \leftarrow rhs$.

```
b=114514
```

```
a=b+1 # 执行完本句之后 a=114515
```

```
b = b+1 # 执行完本句之后 b=114515, a=114515 不变
```

```
a = b+1 # 执行完本句之后 a=114515, b=114516
```

- 在 Python 中, 变量的值的类型可以是任意的. 因为 Python 声明变量的时候没有说明类型.

```
a='Fuhai Zhu teached Advanced Algebra' # a 现在是字符串
```

```
a=1 # a 现在是整数
```

```
a=None # None 是一个关键字, 表示什么都没有.
```

- 如果没有定义就使用了一个变量, 通常就会有如下的报错:

```
print(a+1)
```

```
^
```

```
Traceback (most recent call last):  File "<stdin>", line 1, in <module>
```

```
NameError: name 'n' is not defined
```

(命名错误: 名称'n' 没有定义)

什么是 Traceback? stdin 又是什么? 后面可能会注意到.

可能经常会常用的变量类型: 数字、字符串、列表. 这时候可以参看官方文档 <https://docs.python.org/zh-cn/3/tutorial/introduction.html> 来继续.

控制语句: 判断与循环

定义 1.1.2. (条件判断) 可以使用 if 语句进行条件判断, 一般的, 有如下的形式:

```
if 表达式 1:
    过程 1
elif 表达式 2: # 可以有零个或者多个 elif, 但是 else 后面不能有 elif
    过程 2
else:
    过程 r
```

语义: 它通过逐个计算表达式, 直到发现一个表达式为真, 并且执行使表达式为真的这个过程 (完成后不执行或计算 if 语句的其他部分的判断表达式). 如果所有表达式都为 false, 如果存在 else 下方语句块的过程.

下面我们同样给出注记和例子.

- 什么是真? 什么是假? 我们会在后面探讨. 首先可以认为非 0 数字和 True 是真, 0 和 False 和 None 是假.

```
if "AK":
    print("AK") # 会输出 AK, 这是怎么判断的?(后续会回答)
```

- 可以用逻辑运算符 and(且) or(或) not(非) 进行逻辑表达, 比如

```
zgw = 0
kertz = 1
ak = 1
cmo = 1
if kertz and ak and cmo :
    print(`Zixuan Yuan got full mark in CMO`)
elif zgw and ak and cmo:
    print(`zgw got full mark in CMO`)
else:
    print(`zgw is such a noob`)
# 会输出 Kertz got full mark in CMO, 由于已经找到了一个表达式的值为真的
# 表达式, 所以执行完 print(`Zixuan Yuan got full mark in CMO`) 之后就
# 会跳转到这个语句块的尾部了. 不会执行 print(`zgw is such a noob`).
# (为自己菜爆的数学基础做了一个掩盖 (大雾))
```

- 如果结构不完整, 或者在 else 之后还有 elif, 那么就会出发形如这样的错误:

```
例子 1.py-----
if True:
    print("Err")
```

```

----
File "main.py", line 3
    print("Err")
    ^ IndentationError: expected an indented block
(缩进错误: 我预期有一个带着缩进的语句块, 但是没有)
例子 2.py-----
if False:
    print(1)
else:
    print(2)
elif True:
    print(3)
---
File "main.py", line 5
    elif True:
    ^ SyntaxError: invalid syntax
(语法错误: 无效的语法)

```

定义 1.1.3. (while 循环) 可以使用 if 语句进行条件判断, 一般的, 有如下的形式:

```

while 表达式:
    过程 1
else: # 可以有, 也可以没有
    过程 2

```

语义: 这样反复测试表达式, 如果为真, 则执行**过程 1**; 如果表达式为假 (这可能是第一次测试), 则执行 else 子句的**过程 2**(如果存在的话), 然后循环终止. 在**过程 1**中执行的 **break** 语句会终止循环, 且不执行 else 子句的**过程 2**. 在**过程 1**中执行的 continue 语句跳过**过程 1**的 continue 语句之后的其余部分, 然后立刻回到测试表达式语句.

有了循环, 我们就可以解读这个东西:

```

def InsertionSort(A):
    j=1
    while(j<len(A)):
        key = A[j]
        i = j - 1
        while (i >=0) and (A[i] > key):
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
        j=j+1
    return A
InsertionSort([1,1,4,5,1,4])

```

这个做的事情就和排序成绩类似.

程序的终止

定义 1.1.4. Python 程序的终止可能包含有如下的情况:

- (1) 执行到了最后一条语句, 且没有下一条语句可以执行;
- (2) 程序有没有被处理的异常;
- (3) 通过语句 `exit(0)` 退出.

因此, 我们就得到了最小的可以 (理论上) 执行任何与人类计算能力等价的模型
这些内容看上去十分的平凡, 但是通过一些过程的复合, 我们就能看到更多的魔力.

1.1.6 函数: 整合相似过程

我们可以把相似的过程写在一起, 为了简洁和可维护.

下面, 可以阅读 <https://docs.python.org/zh-cn/3/tutorial/controlflow.html#defining-functions> 的 4.7, 4.8.1-4.8.6 节的内容, 把所有代码是怎么执行的放在 `pythontutor` 里面模拟着看一遍. 文字可以不用看, 但是代码一定要执行一遍.

递归 (Recursion) 过程和栈帧 (Stack Frame)

观察下面的代码, 可能难以想象是怎么执行的:

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

fib(5)
```

像这样用自己调用自己的函数调用通常叫做递归 (recursion). 一个关于递归的有趣定义是:

递归的定义: 如果你没有理解什么是递归, 那么参见递归.

事实上, 我们可以把它放在 `pythontutor` 里面执行一下, 发现如下的规则:

- 原来的程序就像是一张纸, 上面标注着当前执行到的行数;
- 每次函数调用的时候, 就会在一张新的纸片上抄下来调用的内容, 并且代换传进来的参数;
- 把这个内容放在原来纸片上面, 然后从第一行开始执行;
- 执行完的纸片扔掉.

看上去就像是:

- 你在晚自习上看课外书 (执行原来的函数)
- 老师来了, 让你写作业 (函数调用)
- 你把作业叠放在课外书上, 开始做作业 (执行函数)
- 做完作业之后你把作业扔了继续看课外书 (回到原来的函数)

像羽毛球球桶那样, 只能从一个方向插入, 弹出的内容的东西叫做“**栈 (stack)**”, 由于这些内容通常都是一些数据, 由此我们用术语**数据结构** (data structure) 来描述. 能被取出来的那个元素是**栈顶 (top of the stack)**, 在这个可视化工具里面用蓝色标示出来了.

Traceback 就是出错之后, Python 顺着栈一层一层找的结果. Trace 是跟踪, back 是返回, 意思可能就是说堆栈的**回溯 (traceback)**.

1.2 一些小例子

下面我们来看一些基础的例子, 来体会上一节中的一些思考:

片段 1. 热身练习

```
1 def getpercent(chinese, math, english):
2     return (chinese+math+english)/(150+150+150)
3
4 print(getpercent(100, 120, 135))
```

Bonus 思考题

如何判断一个程序的行为? 为了刻画程序的行为, 我们能不能像第一节里面提到的那样, 用一个“模型”来描述它?

宏观来说, 程序就是状态机. 程序的执行就是状态的迁移. 到底有哪些状态呢?

片段 2. 计算钱数之和

```
1 money = [10, 14, 13, 10]
2 int total = 0
3 for i in money:
4     total = total + i
5
6 print(money)
```

这里的 `for i in ...` 只是 `for u in range(len(money)), i=money[u]` 的简称罢了. 对我们的实际程序没有太多的障碍. 还是逃脱不了循环的大框架. 我们在前面说的最小指令集还是起作用的.

片段 3. 求一个数是不是素数

```
1 def isprime(x):
2     flag = 1
3     d=2
4     while d<x:
5         if x%d == 0:
6             flag = 0
7         d = d+1
8     if flag:
9         return 1
10    else:
11        return 0
```

在这个程序片段中, `flag` 代表了什么? 是不是有些像现实世界中的信息传递方式? 因为我们无法跨越循环次序改变程序的行为, 我们只好使用变量 `flag` 来记录, 并且让程序的控制流通过 `flag` 来进行判断与执行.

Bonus 思考题

素数 (prime) 有什么数学性质? 我们是用什么性质来判定性质的? 在我们下达的指令里面, 形式上和数学表达式相似吗?

事实上, 数学上的定义是 $\forall x, \exists p, s.t. p|x$. 我们的程序看上去并不是像数学那样简洁. 其实, 程序设计家族也有其他成员可以写起来比较优美.

Passage 文章**程序设计家族的其他成员: 不只有下达命令**

南京大学 李隼等 节选自《程序分析》教科书

在 IP 中, 指令一个一个给出, 用条件、循环等来控制逻辑 (指令执行的顺序), 同时这些逻辑通过程序变量不断修改程序状态, 最终计算出结果. 我觉得, 尽管 IP 现在都是高级语言了, 但是本质上并没有脱离那种“类似汇编的, 通过读取、写入等指令操作内存数据”的编程方式 (我后面会提及, 这是源于图灵机以及后续冯诺依曼体系结构一脉的历史选择). 国内高等教育中接触的绝大多数编程语言都是 IP 的, 比如 Java、C、C++ 等.

在 FP 中, 逻辑 (用函数来表达) 可以像数据一样抽象起来, 复杂的逻辑 (高阶函数) 可以通过操纵 (传递、调用、返回) 简单的逻辑 (低阶函数) 和数据来表达, 没有了时序与状态, 隐藏了计算的很多细节. 不同的逻辑因为没有被时序和状态耦合在一起, 程序本身模块化更强, 也更利于不同逻辑被并行的处理, 同时避免因并行或并发处理可能带来的程序故障隐患, 这也说明了为什么 FP 语言如 Haskell 在金融等领域 (高并发且需要避免程序并发错误) 受到瞩目.

```
1 (defun quadratic-roots-2 (A B C)
2   (cond ((= A 0) (string "Not a quadratic equation."))
3         (t
4          (let ((D (- (* B B) (* 4 A C))))
```

```

5      (cond ((= D 0) (concatenate 'string "x = " (write-to-string (/ (+ (- B) (sqrt D)) (*
6      (t
7      (values (concatenate 'string "x1 = " (write-to-string (/ (+ (- B) (sqrt D)) (* 2 A)
8      (concatenate 'string "x2 = " (write-to-string (/ (- (- B) (sqrt D)) (* 2 A)
)))))))))

```

LP 抽象的能力就更强了 (用逻辑来表达), 计算细节干脆不见了。把你想表达的逻辑直观表达出来就好了: 如“第三代火影的徒弟”且不是“女性”且“其徒弟也是火影” \Rightarrow “自来也”。“嗯, 学会”与或非, 编程都不怕。如今, 在数据驱动计算日益增加的背景下, LP 中的声明式语言 (Declarative programming language, 如 Datalog) 作为代表开始崭露头角, 在诸多专家领域开拓应用市场。我们这本小书也准备用一章节来教大家如何使用 Datalog 语言编写程序分析器。

```

1 ancestor(A, B) :-
2     parent(A, C),
3     ancestor(C, B).
4

```

片段 4. Perfect 数

```

1 def is_perfect(n):
2     sum=0
3     for i in range(1,n):
4         if n%i==0:
5             sum=sum+i
6     return sum==n

```

其中 `range` 现在可以认为是生成 $[1, n)$ 的列表。并且每一次循环就取列表的下一个元素。比如 `for i in range(1,5)` 每次循环 `i` 的值会是 1 2 3 4。

在这个实例中, 逻辑关系体现的如何?

片段 5. 求 3 和 5 的因数个数

```

1 def multiples_of_3_and_5(n):
2     sum=0
3     for i in range(1, n):
4         if i%3==0 or i%5==0:
5             sum=sum+i
6     return sum

```

可以注意我们的逻辑在 Python 里面是如何表达的? 还有哪些逻辑表达关系? 事实上, 这也是后续离散数学部分要学的命题逻辑—我们需要对于以前的逻辑有一个比较确切的定义。

Bonus 思考题

命题“若 p 则 q ”的否定是什么?

普通的高中毕业生基本是无法回答这个问题的。因为课本的知识完全没有提及类似的问题。这就导致我们的高中数学看上去更像是民科学习的数学。同时轻松地毁掉了高中与大学的衔接

过程.

片段 6. 一定范围内勾股数的个数

```

1 def integer_right_triangles(p):
2     # a^2+b^2=c^2    a+b+c=p    c is the longest
3     count=0
4     for a in range(1,p):
5         for b in range(a,p-a):
6             c=p-a-b
7             if a+b+c==p and a**2+b**2==c**2:
8                 count+=1
9                 print(a,b,c)
10    return count

```

这个例子对于数学的关系好像更加清晰了. 比如直角三角形数对有兴致 $a^2 + b^2 = c^2$. 于是剩下的就比较像自然的“数数”一样了.

片段 7: 递归的力量

```

1 def fib(n):
2     if(n==1):
3         return 1
4     if(n==2):
5         return 1
6     else:
7         return fib(n-1) + \
8             fib(n-2)
9
10 fib(5)

```

这个内容也在前方的例子中有提及, 这样我们自然的就得出了“栈”的定义. 这也是我们这里接触的第一个数据结构—栈.

Idea 启示

计算机高级程序可以由较为低级的程序解释. 这种程序一般而言更加机械, 但是更不利于我们的问题的解答. 这就需要一层一层的抽象叠加起来.

如果自己曾经动手写过一点代码的话, 我们就会发现把代码调试对是一件很不容易的事情. 下面我们给出一些小提示:

(1) **阅读程序的报错信息.** 我们发现很多同学会对于红色的 Syntax Error 如临大敌, 见到就跑. 下面, 我来举一个例子来说明为什么这是对的:

Dialogue 对话

A: 我将会按照一定的规则给出三个数字, 我想让你找出这个规则是什么. 但是你能够获取信息的途径是: 你自己再列举三个数字. 我会告诉你这列数字是不是符合我的规则. 然后你们就可以说出来你们认为的规则是什么.

B: 好的, 明白了.

A: 我说出来的三个数字是 2, 4, 8.

B: 我猜测 16,32,64.

A: 符合我的规则.

B: 那我想规则是 2^n .

A: 其实并不是这样.

为什么会出现这样的情况? 这就是因为没有知道什么东西是“错的”. 请观看真理元素的《你能解决这一问题吗》视频, 思考一下为什么错误也是很重要的. 这位 UP 在 B 站上的官方中文翻译视频链接是<https://www.bilibili.com/video/BV1Hx41157jV>.

(2) 程序出现难以预料的行为时, 在脑子里面模拟执行一遍程序. 告诉自己“程序就是状态机”. 看一看逻辑设计的是不是出错了.

(3) 善于使用调试器. 观察程序在哪一个地方与你预期的执行不相符. 这时候, 往往就意味着可以提问了.

Passage 文章

与其说是学会提问, 倒不如说是学会不提问

南京大学 蒋炎岩

中国科学技术大学 余子豪

节选自《PA 实验手册》

很多同学不多不少都会抱有这样的观点:

我向大佬请教, 大佬告诉我答案, 我就学习了.

但你是否想过, 将来你进入公司, 你的领导让你尝试一个技术方案; 或者是将来你进入学校的课题组, 你的导师让你探索一个新课题. 你可能会觉得: 到时候身边肯定有厉害的同事, 或者有师兄师姐来带我. 但实际情况是, 同事也要完成他的 KPI, 师兄师姐也要做他们自己的课题, 没有人愿意被你一天到晚追着询问, 总有一天没有大佬告诉你答案, 你将要如何完成任务?

如果你觉得自己搞不定, 你很可能缺少独立解决问题的能力.

但幸运的是, 这种能力是可以训练出来的. 你身边的大佬之所以成为了大佬, 是因为他们比你更早地锻炼出独立解决问题的能力: 当你还在向他们请教一个很傻的问题的时候, 他们早就解决过无数个奇葩问题了. 事实上, 你的能力是跟你独立解决问题的投入成正比的, 大佬告诉你答案, 展示的是大佬的能力, 并不是你的能力. 所以, 要锻炼出独立解决问题的能力, 更重要的是端正自己的心态: 你来参加学习, 你就应该尽自己最大努力独立解决遇到的所有问题.

很多问题都可以通过查资料解答. 其中, 有一个很好的途径就是先看一看官方文档. 通常官方文档都有非常详细的解释.

第二章 命题的逻辑

2.1 问题的提出：为什么要研究命题的逻辑

Dialogue 对话

A: 问一个问题: 如果我有一个命题叫做“若 p 则 q ”, 那么这个命题的否定是什么?

B: 我研究这个干什么? 闲着找事情吗?

A: 想一想你学习的极限理论. 如果我希望对于“一个数列的极限存在”这个事情做否定, 你会怎样否定?

B: 数列极限的定义是如果一个数列的极限是 A , 那么就是说 $\forall n > N, \exists |\epsilon| \geq 0, \text{s.t. } |a_n - A| < \epsilon$. 要是否定还是真的不是一件容易的事情啊...

A: 这就是我们学习命题逻辑的原因. 以后我们会遇见成百上千的命题等待我们的操作, 如何从中找到逻辑就至关重要.

事实上, 对于命题逻辑的研究一开始肯定是在数学中接受的. 但是对于数学而言, 我们当中的很多人会发现: 数学仅仅是为了对付高考这样的考试. 那么希望在这一节以及以后的生活中, 慢慢体会数学带给我们的潜移默化的影响.

Bonus 思考题

数学是什么? 仅仅是加减乘除吗? 仅仅是把算法写对吗?

其实这并不是数学的全部.