第 1 讲 · 讲义

计算、机器与语言

课件的每一页，配上讲这一页时说的话。幻灯片是网页原生渲染的，文字可以选中、可以搜索，也可以直接打印。

第 1 页 · 计算、机器与语言

> 这门课的第一讲不教语法。原因是：如果不先弄清楚计算机凭什么能算，后面所有的语法都只是一堆需要背下来的规则。这一讲要回答四个问题——人为什么要造计算机、机器由哪些部分组成、哪些人把「计算」这件事想清楚了、以及这门课为什么选 Python。

什么是计算机

1943 年，美国陆军在报纸上登招聘广告。

职位名称：**computer**。

要求：数学好，有耐心，能长时间专注。

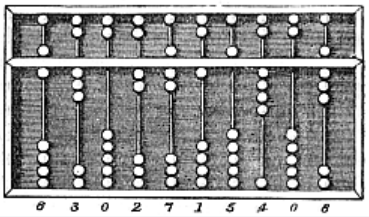
在那个年代，**computer 指的是从事计算工作的人**。

> 1943 年，美国陆军在报纸上招聘 computer。这不是笔误，也不是在招机器。那时候 computer 是一个职业名称，指的是专门从事数值计算的人，要求数学好、有耐心、能长时间保持专注。
>
> 换句话说，中文「计算机」这个词其实翻译错了位置：英文原意是「做计算的人」，后来这个词才转移到机器身上。理解这一点，就能理解整个计算机发展史的动机——人先是自己算，算不动了才去造机器。

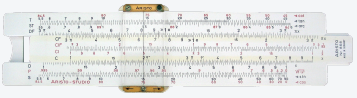
人工计算的时代

天文台和航海需要成千上万页的三角函数表、对数表，这些表由一批被称为 computer 的人手工算出来。

十六世纪的木刻：一边在纸上写数字算，一边拨算板。那时候这是两种**技术路线**之争。



算盘。中国人用了上千年，它把「记住中间结果」这件事交给了珠子。



计算尺。靠对数刻度把乘法变成加法，工程师一直用到 1970 年代——阿波罗登月就是它算的。

人工计算难免出错。**而一页表出错，所有依赖它的航海图都随之出错。**

3

第 3 页 · 人工计算的时代

在机器出现之前，计算是一项人工劳动，而且规模不小。天文台需要三角函数表，航海需要对数表，这些表动辄成千上万页，全部由被称为 computer 的人一格一格算出来。

算盘和计算尺是当时的两类工具。算盘把「记住中间结果」这件事交给珠子；计算尺利用对数刻度，把乘法变成加法——两个数相乘，只要把对应的刻度对齐、读出结果。工程师一直用它到 1970 年代，阿波罗登月的轨道计算就有它的份。

但人工计算有一个无法回避的问题：人会疲劳、会走神、会算错。而这些表是被层层引用的——一页对数表算错，所有依赖它的航海图都跟着错。当时的人很清楚这件事有多要命——待会儿要讲的巴贝奇，造机器的动机正是他受不了数学用表里的错误。所以造机器的动机不是「算得更快」，而是「不会算错」。

让机器来做计算

让机器代替人做计算，这个念头出现得比多数人以为的早得多。

1900 年，一队采海绵的潜水员在希腊**安提基特拉岛**外遇上风暴，躲进背风处下潜，撞见一艘沉了两千年的古罗马货船。

捞上来的文物里有一块锈死的青铜，被当作杂物堆在博物馆。两年后才有人注意到：**那上面有齿。**

此后几十年靠 X 光和 CT 一层层看进去，才认出里面是 **三十多个精密咬合的青铜齿轮**，最小的一个只有十几毫米。

它是一台**天文计算器**：转动手柄输入一个日期，刻度盘就指出那天日月的位置、当晚的月相，还能推算日月食。

制造年代约在**公元前 100 年。用机器代替人计算，这个念头至少有两千年了。**

让机器代替人计算，这个想法出现得比多数人以为的早。

1900 年，一队希腊潜水员出海采海绵，路上遇到风暴，躲到安提基特拉岛背风处下潜避风，结果在四十多米深的海底撞见一艘沉船——古罗马时期的货船，船上装满了雕像和器皿。打捞持续了一年多，其中有一块锈得看不出形状的青铜，当时没人在意，就堆在雅典的博物馆里。

两年后，一位考古学家偶然看到那块青铜上露出的一圈齿，才意识到它不是装饰品。但真正看懂它花了大半个世纪：先是 1970 年代用 X 光透视，后来 2005 年用 CT 一层层扫描，才认出里面有三十多个精密咬合的青铜齿轮，最小的直径只有十几毫米。

它是一台天文计算器。转动手柄输入一个日期，正面的刻度盘会指出那一天太阳和月亮在黄道上的位置、当晚的月相；背面的螺旋刻度能推算日月食，还有一圈专门标注下一届奥林匹克等四年一度的运动会在哪一年举行。

制造年代大约在公元前 100 年。

这一页要留下的就一件事：用机器代替人计算，这个念头至少有两千年了，不是电子时代才冒出来的。至于「有了想法为什么还要等这么久」，等会儿讲巴贝奇的时候有一个证据确凿的例子——他在 1830 年代把通用计算机的结构画了出来，却一台也造不成。

本讲主题

计算机凭什么能算？

本讲不涉及语法。四个问题：计算这件事的由来、机器的结构、三位学者各自的回答，以及本课程选择 Python 的理由。

第 5 页 · 本讲主题

> 这一讲不涉及语法，一行代码都不用写也能听懂。要讲的是四件事：计算这件事的由来、计算机由哪些部件构成、图灵香农哥德尔三个人各自回答了什么问题、以及这门课为什么选 Python。
>
> 如果你今天听不懂任何语法，那是正常的——今天本来就不讲语法。

ROADMAP

今日内容

- **课程说明** —— 规则、成绩、AI 使用政策
- **计算的历史** —— 从算盘到摩尔定律
- **计算机的组成** —— 冯·诺依曼体系，以及「一切都是 0 和 1」的含义
- **计算理论的三个来源** —— 图灵、香农、哥德尔各自回答了什么
- **编程语言** —— 本课程选择 Python 的理由
- **程序的执行过程** —— 以及你的第一个程序

第 6 页 · 今日内容

> 今天分六段。第一段是课程说明：成绩怎么算、AI 能用到什么程度、遇到问题找谁。这一段没有技术内容，但直接关系到你这学期怎么安排时间，尤其是 AI 政策那几页。
>
> 第二段讲计算的历史，从算盘到摩尔定律，跨度两千年。第三段讲计算机由哪些部件组成，以及「一切都是 0 和 1」这句话的确切含义。第四段讲图灵、香农、哥德尔三个人各自回答了什么问题。第五段讲编程语言，说明这门课为什么选 Python。第六段讲你写下的代码是怎么跑起来的，最后写出你的第一个程序。

第 1 段 / 共 6 段

课程说明

> 刚才那个问题先挂着，我们一会儿回来。
>
> 先花二十分钟把这门课的规矩讲清楚：成绩怎么算、AI 能用到什么程度、遇到问题找谁。这一段没有技术内容，但直接决定你这学期怎么安排时间——尤其是 AI 政策那几页，它管着你前七周怎么写作业。

课程组

主讲 黄涛（Dr. Tao Huang）

助教 李晓欧（lixiaoou@sjtu.edu.cn）、苏奕涵（suyihan1@sjtu.edu.cn）

课程网站 taohuang.info/cs1602 讲义、实验、课件、提交、答疑全在上面

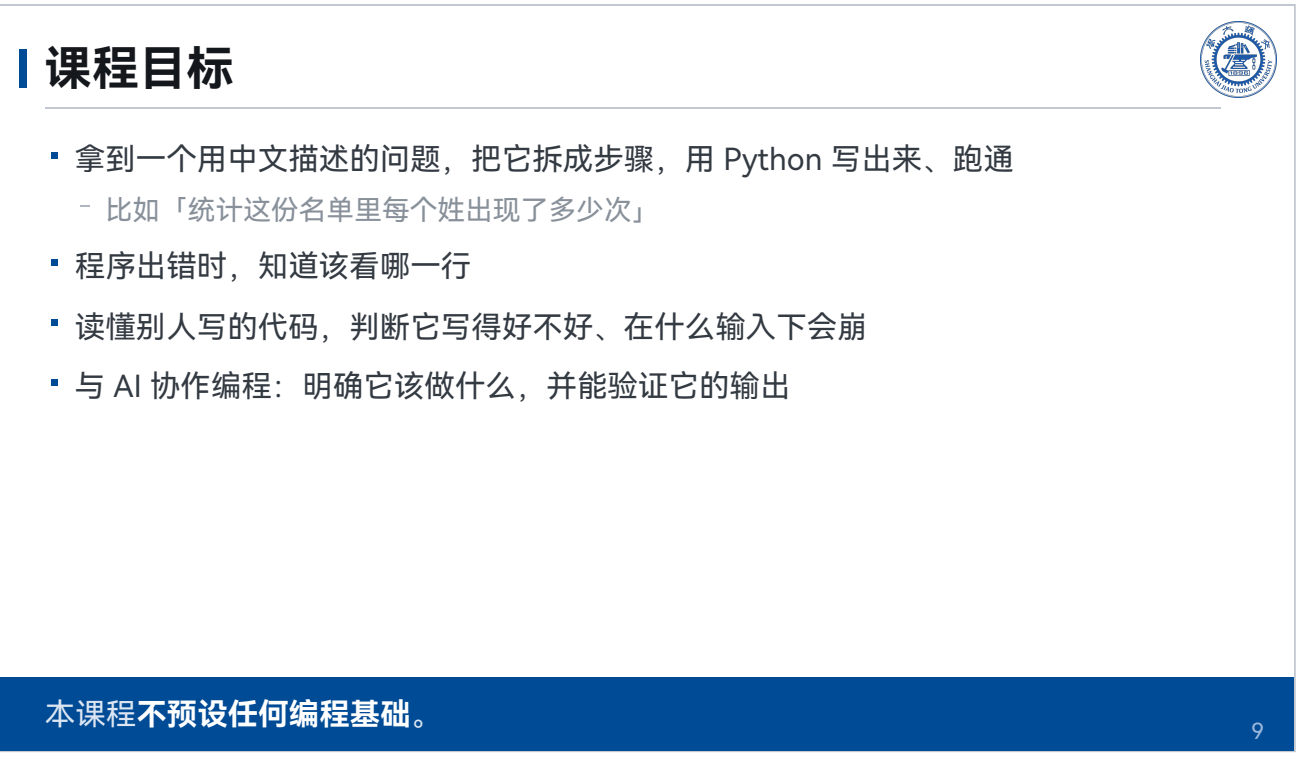
课程微信群 通知和临时答疑走群里，扫左边的码

课程相关的问题，**请优先在群内提出**。

8

第 8 页 · 课程组

> 先把联系方式和入口对齐。
>
> 这门课有两位助教，李晓欧和苏奕涵。作业交给指定的那位，答疑找哪一位都可以。分组第一周在群里公布。
>
> 课程网站 taohuang.info/cs1602 上有全部内容：每一讲的讲义、课件、讲义版课件（幻灯片配讲解）、每周的实验题面，以及实验提交和随堂小测。网站是中英双语的，右上角可以切换。
>
> 微信群用于发通知和临时答疑。课程相关的问题建议优先在群里问：一是回答得快，二是别人多半也有同样的疑问，问一次全班受益。



第 9 页 · 课程目标

先说清楚这门课的终点。十六周之后，你应该能做到四件事。

第一，拿到一个用自然语言描述的问题——比如「统计这份名单里每个姓出现了多少次」——把它拆成若干步骤，用 Python 写出来并跑通。这是最基本的一项。

第二，程序出错时知道从哪里查起。这一项比第一项难，也更重要：写代码的时间里，有相当一部分是在找错，而不是在写。

第三，读懂别人写的代码，并判断它写得好不好、在什么输入下会出问题。

第四，与 AI 协作编程。注意这里说的不是让 AI 替你写，而是你清楚该让它做什么，并且有能力验证它给出的结果。

这门课不预设任何编程基础。如果你此前完全没写过代码，这门课就是按你设计的。

COURSE MAP

课程章节

- **Part I** 计算基础与 Python 入门 —— 写出含输入、判断、循环、函数的完整程序
- **Part II** 数据的组织 —— 为问题选对容器，处理批量数据
- **Part III** 问题求解与 AI 协作 —— 拆解问题；读懂、审查、验证别人写的代码
- **Part IV** 抽象与组织 —— 用类和模块组织中等规模程序
- **Part V** 健壮性与真实世界 —— 评估效率、处理异常、读写真实数据
- **Part VI** Pythonic 与现代实践 —— 写地道的 Python，用 AI 完成一个真实项目

> 十六讲编进六个部分，每个部分收口在一项具体的能力上，而不是一组知识点。
>
> Part I 打基础，学完能写出包含输入、判断、循环和函数的完整程序。Part II 讲数据的组织，重点是「面对一个问题该选哪种容器」。Part III 转向问题求解和 AI 协作，训练的是拆解问题和审查代码的能力。Part IV 讲类和模块，用来组织中等规模的程序。Part V 处理真实世界里的麻烦：效率、异常、文件读写。Part VI 讲怎么写出地道的 Python，并用 AI 完成一个真实项目。

课时安排

- 理论课讲清楚原理
- 实验课形成动手能力
  - 自己编写、自己调试、自己定位错误，这三步无法由他人代替
- 上课**请携带电脑**

写代码的能力是在实验课上形成的。

> 这门课理论 32 学时、实验 32 学时，两者课时相同，但作用不同。
>
> 理论课负责把原理讲清楚，让你知道某件事为什么是这样。但「听懂了」和「写得出来」之间有很大距离——这个距离只能在实验课上跨过去。自己编写、自己遇到报错、自己把错误定位出来，这三步没有任何人可以代替你完成。
>
> 所以理论课请带电脑。片子上的例子可以当场跟着敲一遍，效果比只看好得多。

GRADING

成绩构成

- **小作业 ×5 —— 10%**　上机课的内容，每次覆盖两到四周，合并交一次
- **个人作业 ×3 —— 15%**　汉诺塔 · 素数计数 · 大整数运算，各给两周
- **小组作业 ×1 —— 25%**　两人一组，做一个简化版 NumPy。**只提交，不答辩**
- **期末考试（闭卷） —— 50%**

期末占 50%；**另一半分散在整个学期里**，每周都有机会拿到，也每周都可能漏掉。

成绩由四部分构成。

平时的作业分三类，这是你这学期大部分时间要花的地方。第一类是小作业，一共五次，内容就是上机课上做的东西；一次小作业覆盖两到四周，合并交一份，占 10%。第二类是个人作业，三次，每次给两周：汉诺塔、素数计数、大整数运算，合计 15%。第三类是小组作业，两人一组做一个简化版的 NumPy，占 25%——注意它只提交，不答辩。

最后是期末闭卷考试，占一半。

期末占一半，意味着期末不能靠突击。但反过来看，另外 50% 分散在整个学期里，每周都有机会拿到——这部分是过程分，只要按时动手就不会丢。往年丢分最多的恰恰是这一半，因为它不像考试那样有明确的截止压力。

AI 使用政策

这是这门课最重要的一条规则

第 13 页 · AI 使用政策

> 接下来这一条是这门课最重要的规则，请认真听。
>
> 它关系到你前七周怎么写作业，也关系到违反之后怎么处理。下一页是规则本身，再下一页解释为什么要有这条规则——第二页比第一页更重要，因为不理解理由的规则很容易被当成形式主义。

AI POLICY

AI 政策的三个阶段

- **第 1–7 周：不允许用 AI 生成或补全代码。**
  - 请关闭编辑器里的 AI 补全功能
  - 可以用 AI 问概念、查文档、解释报错 —— 但不能让它替你写代码
- **第 8 周起：允许使用，但每次提交要附一段声明。**
  - 说明你用了什么工具、它生成了哪部分、你如何验证它是对的
- **第 15–16 周（团队项目）：完全开放。**

完整规则在课程网站的「AI 政策」页。

第 14 页 · AI 政策的三个阶段

AI 政策分三个阶段。

第 1 到 7 周，不允许使用 AI 生成或补全代码。请注意「补全」也在其中——很多人并非有意违规，而是根本没意识到编辑器里的 AI 补全一直开着。装环境的时候就把它关掉，比事后再关可靠。这一阶段仍然可以用 AI 问概念、查文档、解释报错信息，只是不能让它替你写代码。

第 8 周起允许使用，但每次提交要附一段声明，说明用了什么工具、它生成了哪一部分、你用什么方法验证它是对的。写这段声明本身就是训练。

第 15 到 16 周的团队项目完全开放，不再限制。

完整规则在课程网站的「AI 政策」页面。

前七周禁用 AI 的理由

- 前七周你要建立的是两样东西：**代码直觉**和**调试能力**
  - 看到一段代码，能预判它会输出什么
  - 看到一个报错，能猜到问题在哪一行
- 这两项能力没有捷径，只能通过自己编写、出错、排查获得
- 由 AI 代写，代码可以运行，**但你无法判断它在什么情况下会出错**

第八周之后能否用好 AI，**取决于这一阶段积累的判断力**。

15

第 15 页 · 前七周禁用 AI 的理由

前七周为什么要禁，这一点需要解释清楚。

这一阶段你要建立两样东西：代码直觉和调试能力。代码直觉是指看到一段代码，能大致预判它会输出什么；调试能力是指看到一个报错，能猜到问题出在哪一行。

这两样都没有捷径，只能通过自己编写、自己出错、自己排查获得——就像学游泳不能靠看别人游。

如果这一阶段把代码交给 AI 写，你会得到一批能运行的代码，但不会获得判断这些代码对错的能力。而 AI 经常写出看起来完全合理、却在边界情况下失败的代码：空列表、除零、下标越界、输入格式和预期不一致。没有前七周的积累，这些问题你看不出来。

这条政策的目的

前七周的限制，目的是让第八周之后的你具备判断能力

16

第 16 页 · 这条政策的目的

> 把这条规则的目的说明白：限制的不是使用 AI 本身，而是把使用的时间点往后推，先让你具备判断能力。
>
> 第八周之后，你和 AI 协作时最值钱的东西就是这个判断力——知道它给的东西哪里可能不对，并且知道怎么验证。所以前七周的限制，收益兑现在后半个学期。

遇到问题的处理顺序

- **读报错信息。** 它通常直接告诉你哪一行、什么问题
  - 常见的问题是看到报错就跳过，并没有真正去读它
- **查官方文档。** docs.python.org 有完整中文版
- **搜索。** 把报错里的关键部分贴进搜索引擎，大概率有人遇到过
- **问同学、问助教。** 课程群里有问必答
- **问 AI** —— 在政策允许的范围内

> 遇到问题时按这个顺序处理，顺序是有讲究的。
>
> 第一步读报错信息。Python 的报错通常直接指出哪一行、什么类型的问题，有时甚至给出修改建议。新生最常见的问题是看到红色文字就跳过，根本没有读它。
>
> 第二步查官方文档，docs.python.org 有完整的中文版。
>
> 第三步搜索：把报错信息中的关键部分贴进搜索引擎，你遇到的问题大概率别人也遇到过。
>
> 第四步问同学和助教，课程群里有问必答。
>
> 第五步才是问 AI，并且要在政策允许的范围内。
>
> 前三步都是你能自己完成的，练的正是自学能力。如果每次都直接跳到最后一步，学期末你会发现自己什么都没学会。

有效提问的三个要素

- **你想做什么** —— 目标是什么
- **你写了什么** —— 贴出代码，而不是复述代码
- **发生了什么** —— 贴出完整的报错信息

「我的代码不对，怎么办」——**这样的提问无法得到有效回答**。

18

第 18 页 · 有效提问的三个要素

> 提问的方式直接决定你能得到什么样的回答。一个可回答的问题包含三部分：你想做什么、你写了什么、发生了什么。
>
> 第二部分要贴出代码，而不是复述代码——「我用了一个循环」和实际那五行代码提供的信息量完全不同。
>
> 第三部分要贴出完整的报错，从 Traceback 那一行开始一直到最后。只说「报错了」等于没说。
>
> 「我的代码不对，怎么办」这样的提问，任何人都无法回答，AI 也不能。这个习惯对问助教、问 AI、以及将来在工作中提问同样有效。

TAOHUANG.INFO/CS1602

课程网站

- **讲义** —— 每一讲一篇，比幻灯片详细得多，是这门课的教材
- **课件** —— 本课使用的幻灯片，可逐页翻阅，也可下载 PDF
- **讲义版课件** —— 每页幻灯片配上对应的讲解，适合课后阅读
- **实验** —— 每周的题面和要求
- **实验提交** —— 交作业、看判题结果
- **随堂小测** —— 本讲稍后会用到

中英文两版，右上角切换。讲义和实验都是双语的。

第 19 页 · 课程网站

课程网站上有六类内容。

讲义是每一讲一篇的长文，比幻灯片详细得多，实际上是这门课的教材，考试范围以它为准。课件就是现在投影的这些幻灯片，可以逐页翻阅，也能下载 PDF。讲义版课件把每一页幻灯片和对应的讲解排在一起，适合课后复习——你现在读到的这段文字就是它。

实验页面放每周的题面和要求，实验提交页面用来交作业和查看判题结果。随堂小测就是本讲稍后会用到的那个。

全站中英双语，右上角切换。

这门课的两个入口

两个都要收藏。日常在课程网站，**Canvas 是最终交作业和看成绩的地方**。

课程网站 `taohuang.info/cs1602` —— 讲义、课件、实验、提交判题、随堂小测，**平时都在这里**

Canvas `oc.sjtu.edu.cn/courses/97484` —— 学期末把评测报告交到这里，成绩和教务通知也走这边

平时的题目和判分在课程网站；**学期末下载一份评测报告，交到 Canvas** —— 源码已经嵌在报告里，不用另外交 .py。

20

第 20 页 · 这门课的两个入口

这门课有两个入口，两个都要收藏。

左边是课程网站，taohuang.info/cs1602。平时的东西都在这里：讲义、课件、每周的实验题面、实验提交和判分、随堂小测。你这学期打开得最多的就是它。

右边是 Canvas，学校的教学系统，这门课的地址是 oc.sjtu.edu.cn 斜杠 courses 斜杠 97484。它管两件事：学期末你要把评测报告交到这里，以及成绩和教务那一侧的通知走这边。

分工是这样的：你在课程网站上做题、提交、当场看判分，学期末从平台下载一份评测报告——你提交过的源码已经嵌在报告里了——把那一个文件交到 Canvas，不用另外交 .py。

现在就扫，两个都存进浏览器书签。

CHECKLIST

本节课后待办

- **加课程群**（二维码在前面那页）
- **课程网站和 Canvas 都加进书签**（上一页两个码）
- **本周实验课请携带电脑** —— 环境安装是实验课的第一项内容

环境安装安排在 **Lab 1**，届时会逐步演示。

> 下课前请完成三件事：加入课程群、把课程网站加入书签、以及记住本周实验课要带电脑。
>
> Python 和 VS Code 不用现在安装。环境安装是实验课的第一项内容，Lab 1 会带着你一步一步做完，遇到问题助教在现场。自己在宿舍折腾到半夜却装不上，是每年第一周最常见的情况，没有必要。

第 22 页 · 随堂小测 · 第 1 轮

> 这是今天的第一轮小测，两道题。扫幻灯片上的二维码，在手机上作答。作答是匿名的，我只看得到全班的分布，看不到是谁选的。
>
> 第一题是问卷，不是考题，没有对错。它有两个用途：一是让我看到全班的起点分布，好决定前几讲的节奏；二是让你熟悉这套流程，后面每一讲都要用。看到分布之后你会发现，完全没写过代码的人比想象中多得多——这门课就是按零基础设计的。
>
> 第二题考刚才那条 AI 政策，这是今天唯一一道必须答对的题。题干很具体：第 3 周写作业时打开了编辑器的 AI 补全，它替你补出了一个循环，算不算违规。如果班上有相当比例选错，说明规则那两页我讲得不够清楚，我会再讲一遍。

第 23 页 · 回到开头那个问题

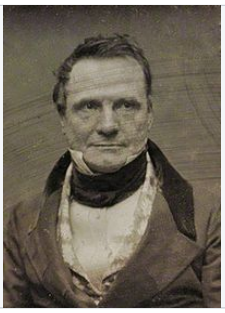
> 规矩讲完了，回到开头那个问题。
>
> 1943 年美国陆军招的 computer 是人。那么今天摆在你面前的这台机器，凭什么能算?
>
> 接下来的四段就是在回答这一个问题，而且是一条线下来的：人为什么算不动 → 机器怎么接手 → 接手之后它内部只有 0 和 1 → 这串 0 和 1 凭什么有意义 → 你怎么把自己的意图变成它。
>
> 每一段结束我都会停一下，说清楚这一段确定了什么。你要是中间走神了，听那一句就能跟回来。

第 2 段 / 共 6 段

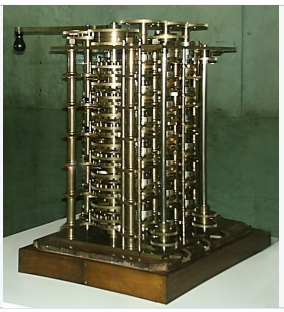
计算的历史

> 回到开头。那三页说的是：计算本来是人干的活，而人自己算有两个过不去的坎——算不动，而且会算错。
>
> 所以这一段要问的是：想让机器接手，当年卡在哪儿，后来是怎么过去的。从十九世纪的齿轮讲到今天的芯片，两百年，但你会看到真正起作用的其实只有一件事。

十九世纪的机械计算机



查尔斯·巴贝奇（Charles Babbage, 1791–1871）。他受不了对数表里的错误，决定造一台不会错的机器。



差分机。用齿轮做加法，摇手柄出结果。后来他又设计了更野心的**分析机**。

分析机已经有了处理器、存储器和输入输出的雏形，用**打孔卡**编程。

十九世纪出现了第一次认真的尝试。

查尔斯·巴贝奇是英国数学家。他的动机不是速度，而是错误——他受不了当时数学用表里层出不穷的印刷和计算错误，决定造一台不会算错的机器。

差分机是他的第一个设计，用一组齿轮做加法，摇动手柄就能逐项算出多项式的值。之后他又设计了野心大得多的分析机：它有专门做运算的部件、存放数据的部件，还能通过打孔卡输入指令。

换成今天的说法，分析机已经具备了运算器、存储器和输入输出——也就是说，一台通用计算机的完整设计在 1830 年代就画出来了，比第一台电子计算机早了一百多年。

分析机的结局

当时的机械加工精度跟不上设计

但图纸留了下来。巴贝奇常被称为「计算机之父」，靠的就是这套图纸。

> 但分析机一台也没造出来。原因不在设计，在制造：那个年代的机械加工精度达不到要求，几千个齿轮做不到既精确又不卡住。巴贝奇为此耗尽了资金和大半生。
>
> 留下来的是图纸。巴贝奇被称为「计算机之父」，依据就是这套图纸——在计算这件事上，把结构想清楚本身就是最难的部分。
>
> 顺带一提，1991 年伦敦科学博物馆按巴贝奇的原图纸造出了一台差分机，只用当时能达到的工艺，结果它能正常工作。所以设计本身没有问题。

第一个计算机程序

她为一台还不存在的机器，写下了一组可以执行的指令。

阿达·洛芙莱斯（Ada Lovelace, 1815–1852）。诗人拜伦的女儿，数学家。

1843 年，她把一篇介绍分析机的法文文章译成英文，又在译文后面附了一组注释——**篇幅是原文的三倍**。

其中的「**注释 G**」给出了用分析机计算**伯努利数**的完整步骤：怎么循环、中间结果存在哪一格、什么时候再取出来。

她面对的是一台**从未存在过的机器**——没有硬件可以试，全部推演都在纸上完成。

这被认为是**世界上第一个计算机程序**——比第一台真正的计算机早了整整一百年。

27

第 27 页 · 第一个计算机程序

> 阿达·洛芙莱斯是诗人拜伦的女儿，也是一位数学家。她在翻译一篇介绍分析机的文章时，加了一组比原文长两倍多的注释。其中一条注释里，她写下了用分析机计算伯努利数的完整步骤——包括如何循环、如何在中间存放结果。
>
> 这被普遍认为是世界上第一个计算机程序。值得注意的是，她面对的是一台从未存在过的机器：没有硬件可以试，所有的推演都在纸上完成。这件事和今天写代码是同一种智力活动——在脑子里模拟一台机器会怎么一步步执行你写下的东西。

洛芙莱斯的洞见

- 巴贝奇想的是：这台机器能**算数**
- 洛芙莱斯想的是：只要能把别的东西编码成数字 ——
  - 比如音符 ——
- **它就能处理任何东西**

今天的照片、音乐与文本都是数字。**这一点她在 1843 年已经指出。**

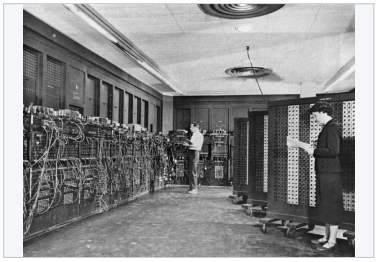
28

第 28 页 · 洛芙莱斯的洞见

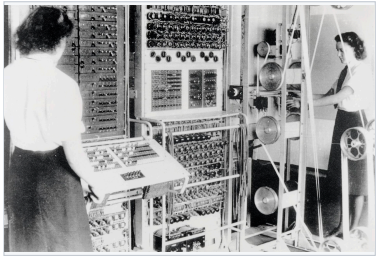
> 洛芙莱斯还看到了一件巴贝奇没有看到的事。
>
> 巴贝奇想的是：这台机器能算数。洛芙莱斯想的是：如果能把别的东西编码成数字，那么这台机器就能处理那样东西。她在注释里举的例子是音乐——只要把音高和时值编成数字，机器就可以用来作曲。
>
> 今天你手机里的照片是一组数字，音乐是一组数字，聊天记录也是一组数字。整个数字世界建立在这条判断上，而它写于 1843 年。

开关器件的三次更替

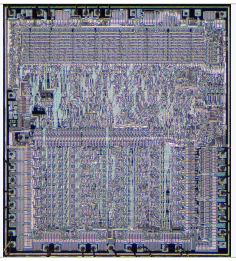
从继电器到电子管到晶体管，换的其实是同一样东西：**开关**。



电子管（1940s）。ENIAC 有 18000 支，占满一个大厅，开机时费城的灯会暗一下。



ENIAC 的操作员。**当年的程序员大多是女性**——那时候「编程」被当成文书工作。



晶体管 → 集成电路（1947 至今）。今天这么大一块上有几百亿个开关。

三次更替的对象始终是开关。**每一次都更小、更快、更省电。**

29

第 29 页 · 开关器件的三次更替

机器真正开始计算，靠的是三种不同的开关。

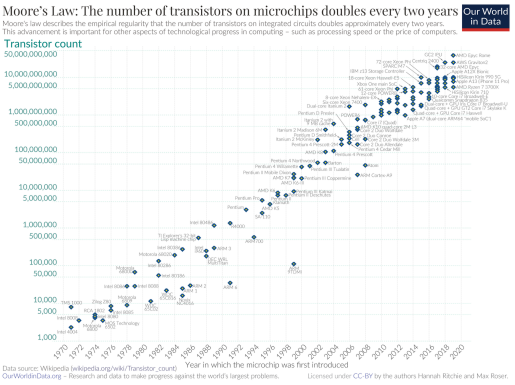
最早是继电器，用电磁铁把机械触点吸合，靠触点通断表示 0 和 1。接着是电子管，没有机械部件，速度快得多。ENIAC 用了约 18000 支电子管，占满一个大厅，功耗巨大，而且电子管会烧坏——平均每隔一两天就要换一支。

1947 年贝尔实验室发明晶体管，这是这条线上最关键的一步：体积小、不发热、不烧坏、可以批量制造，进而可以集成到一块硅片上。1970 年的 MOS 6502 处理器上有 3500 个晶体管，今天一枚手机芯片上有几百亿个。

顺便说一句 ENIAC 那张照片：当年负责给机器编程的六个人全是女性。那时候「编程」被当作文书性的工作，地位不高，后来这个行业的性别构成才发生了变化。

摩尔定律

1965 年，英特尔创始人之一戈登·摩尔注意到一个规律。



集成电路上的晶体管数目随年份变化。纵轴是对数刻度 ——一条直线意味着指数增长。

集成电路上能容纳的晶体管数目，大约每 18 个月翻一番。

30

第 30 页 · 摩尔定律

1965 年，英特尔创始人之一戈登·摩尔在一篇文章里指出：集成电路上能容纳的晶体管数目，大约每 18 到 24 个月翻一番。

看这张图时请注意纵轴：它是对数刻度，每一格代表十倍。在对数坐标上，一条直线意味着指数增长。这是你在这门课里第一次遇到对数坐标，以后讲复杂度时还会遇到。

指数增长的直观感受很不可靠。翻一番听起来不多，但连续翻十次就是一千倍，翻二十次就是一百万倍。从 1970 年到今天，晶体管数目正好增长了大约一千万倍。

摩尔定律的三种读法

- **性能** —— 同样的价格，18 个月后能买到快一倍的机器
- **价格** —— 同样的性能，18 个月后价格减半
- **尺寸** —— 同样的功能，18 个月后体积缩小一半

它不是物理定律，是**经验观察**，而且近年已经明显放缓。

> 摩尔定律有三种等价的读法，三种都要知道。
>
> 第一种是性能：花同样的钱，18 个月后能买到快一倍的机器。第二种是价格：要同样的性能，18 个月后只要一半的钱。第三种是尺寸：实现同样的功能，18 个月后体积缩小一半。手机、智能手表、耳机里的芯片能存在，靠的是第三种。
>
> 还要说清楚它是什么：这不是物理定律，而是一个经验观察，并且已经明显放缓——晶体管的尺寸正在逼近原子尺度，3 纳米大约相当于十几个硅原子排成一列的宽度，再小就要面对量子效应。

算力增长的影响

很多二十年前「算不动」的问题，今天可以硬算

你手里这台手机，算力超过 1990 年代的超级计算机。很多学科现在的做法是：先建模型，让机器算，最后才回实验室验证。

> 这条曲线对你的直接影响是：很多二十年前算不动的问题，今天可以直接硬算。
>
> 你手里这台手机的算力超过 1990 年代的超级计算机。气候模型、蛋白质结构预测、大语言模型，它们背后的数学并不都是新的，变的是算得动了。
>
> 所以很多学科现在的工作方式是：先建立数学模型，让机器算出结果，最后才回到实验室去验证。计算成了理论和现实之间的一环。

人工智能的两个里程碑

1997，深蓝击败国际象棋世界冠军卡斯帕罗夫。路子是穷举——把所有可能都算一遍。

2016，AlphaGo 击败李世石。围棋的搜索空间太大，穷举走不通，它换了一条路：**学习**。

两者相隔 19 年。当前的模型已不限于博弈，**也能编写代码**。

两个标志性的时刻，中间隔了 19 年。

1997 年，IBM 的深蓝在国际象棋上击败世界冠军卡斯帕罗夫。深蓝走的是穷举路线：每一步尽可能多地往后推算，靠算力压过人。

2016 年，AlphaGo 击败李世石。围棋不能用同样的办法——棋盘上合法局面的数量远远超过可穷举的范围。AlphaGo 换了一条路：不再试图算尽所有可能，而是从大量棋局中学习「哪一步看起来更好」。

隔了 19 年，是因为这两条路差别很大。而现在的模型不只会下棋，也能编写代码——你们这一届入学时，正好赶上这一段。这门课的第三部分和第六部分会正面处理这件事。

第 34 页 · 两百年，换的其实是同一样东西

> 把这一段收一下。
>
> 从巴贝奇到今天，两百年里换过四种东西：齿轮、继电器、电子管、晶体管。齿轮那一版没成，卡在加工精度上；后面三种都是电控的开关，一次比一次小、比一次快、比一次省电，摩尔定律说的就是这条线。
>
> 但请注意：它们干的是**同一件事**——通，或者断。两百年的技术进步，全部花在「把开关做得更小更快」上，开关本身的性质一点没变。
>
> 所以下一段只剩一个问题：只有通和断两个状态，怎么装得下一整台计算机？

第 3 段 / 共 6 段

计算机的组成

> 上一段收在一个很窄的结论上：机器能算，靠的是开关，而开关只有通和断两个状态。
>
> 这一段就从这儿往下走：只有两个状态，怎么装得下一整台计算机。答案有两层——一层是结构，五个部件怎么摆；一层是含义，一串 0 和 1 到底算什么。第二层是今天最要紧的一页。

冯·诺依曼与 EDVAC 报告

约翰·冯·诺依曼（John von Neumann, 1903–1957）。匈牙利裔数学家，在数学、物理、经济学上都留下了奠基性的工作。

1945 年 6 月，他为 EDVAC 计算机项目写了一份报告草稿，在里面描述了一种计算机的组织方式：**五个部件，而且程序和数据放在同一个地方。**

这份草稿本来只在项目组内部传阅，却被油印散了出去。此后造机器的人，基本都照着它建。

需要说明：这个结构不是他一个人的功劳，ENIAC 团队的埃克特和莫奇利做了关键工作——只是那份报告上**署了他一个人的名字**。

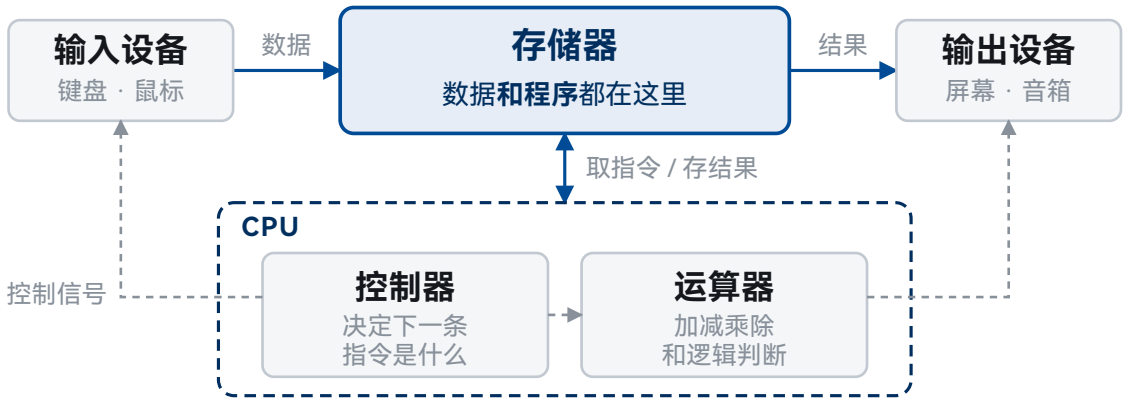
今天你能碰到的几乎所有计算机 —— 手机、笔记本、服务器、路由器 ——**都是这个结构的变体**。

36

第 36 页 · 冯·诺依曼与 EDVAC 报告

> 1945 年，约翰·冯·诺依曼在一份关于 EDVAC 的报告草稿里，描述了一种计算机的组织方式。这份报告流传开来之后，后来的机器基本都按这个结构建造。
>
> 今天你能接触到的几乎所有计算机——手机、笔记本、服务器、路由器、汽车里的控制器——都是这个结构的变体。八十年里变的是速度、体积和功耗，结构本身没有变。
>
> 需要说明的是，这个结构并非他一人的功劳，ENIAC 团队的埃克特和莫奇利也做了关键工作，但这份报告的署名方式让这个名字流传了下来。

冯·诺依曼体系的五个部件



实线是数据，虚线是控制。 控制器指挥所有人，但它自己不搬数据 —— 这是「谁在干活」和「谁说了算」的分工。

对着自己的笔记本，这五样都能一一指出来。**注意存储器那一格：它存的不只是数据。**

37

第 37 页 · 冯·诺依曼体系的五个部件

冯诺依曼体系由五个部件构成，图上这么看最清楚。

上面一排是数据的去向：输入设备把外界的信息送进来，进存储器；算完之后从存储器出去，交给输出设备。键盘鼠标摄像头属于前者，屏幕音箱打印机属于后者。

下面这个虚线框是 CPU。请注意：五个部件不是五个平铺的盒子——运算器和控制器这两个是装在 CPU 里的。运算器做加减乘除和逻辑判断，控制器决定下一条执行哪条指令。

中间那条双向箭头是 CPU 和存储器之间的来回：取指令、取数据下去，算完的结果再存回来。你以后会反复听到「内存带宽」这个词，说的就是这条线有多粗。

虚线画的是控制信号。控制器指挥所有其他部件，但它自己不搬数据——「谁在干活」和「谁说了算」是分开的。

最后看存储器那一格：里面写的是「数据和程序都在这里」。这一句是下一页的关键，也是整个体系里最要紧的一点。

存储程序思想

程序和数据存在同一个地方，用同样的方式表示

这叫「存储程序」思想。后果是：机器不用重新接线就能换任务，只要把另一段程序读进存储器。而且——程序本身也可以被当作数据来处理。

> 这个设计里最关键的一点是：程序和数据存放在同一个地方，并且用同样的方式表示——都是一串 0 和 1。这叫「存储程序」思想。
>
> 它带来两个后果。第一，机器换任务不需要重新接线，只要把另一段程序读进存储器就行。ENIAC 换一个计算任务要重新插接线路，有时要花好几天；存储程序把这件事变成了「打开另一个文件」。
>
> 第二个后果更深远：既然程序也是数据，那么程序就可以被另一个程序读取和处理。编译器、解释器、代码格式化工具、乃至今天能写代码的语言模型，根子都在这一条上。

二进制与进制表示

我们把它们记作 1 和 0。一个 0 或 1 叫一个 **bit**（位），八个 bit 叫一个 **byte**（字节）。

进制

```
1  n = 2026
2
3  print("十进制:", n)
4  print("二进制:", bin(n))
5  print("八进制:", oct(n))
6  print("十六进制:", hex(n))
```

output

```
十进制: 2026
二进制: 0b11111101010
八进制: 0o3752
十六进制: 0x7ea
```

`0b` `0o` `0x` 是前缀，告诉你后面这串数字按几进制读。

> 存储器里只有两种状态：高电压和低电压，我们把它们记作 1 和 0。一个 0 或 1 叫一个 bit（位），八个 bit 叫一个 byte（字节）。
>
> 同一个数可以用不同的进制表示。片子上这段代码把 2026 分别用十进制、二进制、八进制、十六进制打印出来。输出里的 0b、0o、0x 是前缀，用来标明后面这串数字该按几进制读——没有前缀的话，1010 到底是十进制的一千零一十还是二进制的十，就无法判断。
>
> 你可以把 2026 换成自己的学号试一试，看看它在二进制下有多长。

二进制序列的解释

同一串 01000001

```
1  bits = 0b01000001
2
3  print("当作整数:", bits)
4  print("当作字符:", chr(bits))
5  print("当作红色深浅:", bits / 255)
```

output

```
当作整数: 65
当作字符: A
当作红色深浅: 0.2549019607843137
```

二进制序列本身没有意义，意义来自约定。记住这一句 —— 第 5 段讲的「语言」，就是人和机器之间的一层约定。

40

第 40 页 · 二进制序列的解释

这一页是今天最重要的一个观念。

内存里存着 01000001 这八个位。它到底是什么？当作整数读，它是 65；当作 ASCII 字符读，它是字母 A；当作某种图像格式里的一个灰度值读，它是 65/255 的深浅；当作机器指令读，它可能是某条 CPU 指令。

所以：二进制序列本身没有意义，意义来自约定。那计算机怎么知道该按哪一种读？它不知道——是程序决定的。同一段内存，你用 int 去读它就是整数，用 str 去读它就是文字。

这就是为什么后面会有「类型」这个概念，也是为什么把类型搞错时程序不会报错、只会给出莫名其妙的结果。第 2 讲会详细展开。

操作系统

- 操作系统管理硬件资源，并向其他程序提供**统一的服务**：
  - 你写 `open("data.txt")` 时，不需要知道硬盘的磁头在哪个磁道
  - 你同时开着浏览器和音乐播放器，是操作系统在决定 CPU 现在给谁用
  - 你按下键盘，是操作系统把这个事件送到当前窗口

Windows、macOS、Linux、iOS、Android 都是操作系统。**第 8 次实验会带你接触 Linux 命令行**——服务器几乎都跑在上面。

41

第 41 页 · 操作系统

> 硬件是一堆能通电的元件，让它们做有用的事需要软件。软件里最重要的一个是操作系统。
>
> 操作系统管理硬件资源，并且向其他程序提供统一的服务。举三个你每天都在用的例子：你写 open("data.txt") 打开一个文件时，不需要知道这个文件在硬盘的哪个位置、磁头要移动到哪一道；你同时开着浏览器和音乐播放器时，是操作系统在决定 CPU 这一刻给谁用；你按下一个键时，是操作系统把这个事件送到当前活动的窗口。
>
> 这三件事的共同点是「屏蔽细节」。Windows、macOS、Linux、iOS、Android 都是操作系统。这门课第 8 次实验会带你接触 Linux 命令行，因为服务器和科学计算集群几乎都运行在 Linux 上。

第 4 段 / 共 6 段

计算理论的三个来源

> 到这里为止，讲的都是机器怎么一步步造出来的。现在换一个问题：它能算什么，又有什么是它算不了的。
>
> 请特别注意这一段的年份。哥德尔 1931，图灵 1936，香农 1937——第一台电子计算机 ENIAC 要到 1946 年才运转，冯诺依曼那份报告是 1945 年。也就是说，**这三个人是在还没有计算机的时候，把「计算」这件事想清楚的**。
>
> 所以前面那个跳跃是有意的：我们先看机器怎么造，再回到更早的地方，看造它之前得先想明白什么。

图灵机

1936 年，他提出一个极简的假想机器：一条无限长的纸带，一个读写头，一张状态转移表。

阿兰·图灵（Alan Turing, 1912–1954）

它每次只能读一格、写一格，并左移或右移一格。**仅此而已。**

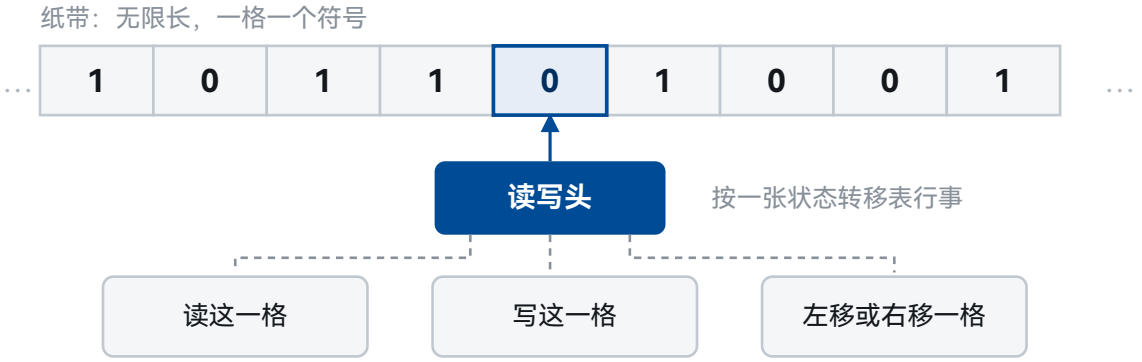
1936 年，阿兰·图灵提出了一个极简的假想机器。

它由三部分组成：一条无限长的纸带，纸带被分成一格一格，每格能写一个符号；一个读写头，停在某一格上；一张状态转移表，规定「在状态 X 读到符号 Y 时，写下什么、往哪边移一格、切换到哪个状态」。

它每次只能读一格、写一格，然后左移或右移一格。就这些，没有别的。第一次看到这个模型的人通常会觉得它太简陋，干不了什么事。

图灵机由三样东西组成

一条无限长的纸带、一个读写头、一张状态转移表。



它每一步只能读一格、写一格、左右挪一格。仅此而已。第一次看到的人通常觉得它太简陋，干不了什么事。

44

第 44 页 · 图灵机由三样东西组成

> 把图灵机画出来看。
>
> 它由三部分组成。第一是一条无限长的纸带，分成一格一格，每格能写一个符号——图上写的是 0 和 1，实际上用什么符号都行。第二是一个读写头，停在某一格上，就是图上那个蓝块。第三是一张状态转移表，规定「在状态 X 读到符号 Y 时，写下什么、往哪边移一格、切换到哪个状态」。
>
> 它每一步能做的事，就是底下那三个框：读这一格、写这一格、左移或者右移一格。就这些，没有别的。没有内存、没有寄存器、没有乘法指令。
>
> 第一次看到这个模型的人通常会觉得它太简陋了，这么个东西能干什么。下一页就是答案。

可计算性的定义

任何能被机械地计算出来的东西，都能用它算出来

「可计算」自此有了精确的定义。

> 但图灵证明了：任何能够被机械地、按明确步骤计算出来的东西，都能用这台机器算出来。
>
> 这个结论的分量在于，它给「计算」这个此前含糊的词一个精确定义：所谓可计算，就是存在一台图灵机能算出它。
>
> 由此得到一个反直觉的推论：你手里这台手机，在「能算什么」这个问题上并不比那条纸带更强——它只是快得多。凡是手机能算的，图灵机给足时间也能算；凡是图灵机算不了的，再快的机器也算不了。

不可计算问题与停机问题

- **有些问题任何计算机都解不了** —— 不是算得慢，是根本不存在算法
- 最著名的是**停机问题**：
  - 不存在一个程序，能判断任意给定的程序是否会陷入死循环

推论：既然连「这个程序会不会死循环」都无法自动判断，那么**「这段代码有没有 bug」当然也无法自动判断**。

同一套理论也划出了边界：有些问题任何计算机都解不了。注意这里说的不是「算得太慢」，而是根本不存在算法。

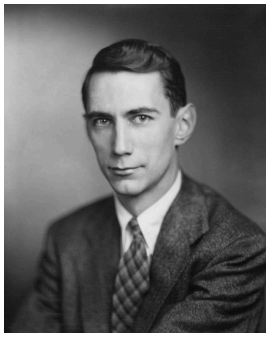
最著名的是停机问题：不存在这样一个程序，它能对任意给定的程序判断出「这个程序会不会陷入死循环」。这个结论是严格证明出来的，不是「目前还没找到办法」。

对写程序的人来说，它有一个非常实际的推论：既然连「会不会死循环」都无法自动判断，那么「这段代码有没有 bug」当然也无法自动判断——后者比前者更难。

所以测试能增加信心，但永远不能证明程序没有错误。这句话在第 11 讲讲测试时还会用到。

顺带一提，图灵在二战期间参与破译德军的 Enigma 密码，对战争进程有实质影响。计算机科学的最高奖以他命名。

香农与信息的度量



克劳德·香农（Claude Shannon, 1916–2001）。美国数学家、电气工程师。

他在国内的名气不如图灵，但今天技术的形态更直接地由他决定：你天天说的「**比特**」，第一次出现在印刷品上就是他 1948 年那篇论文里。

他也是出了名的爱玩：骑独轮车穿过贝尔实验室的走廊，做过一台唯一功能是伸出手把自己关掉的机器，还做过世界上第一只会走迷宫、并且记得住路的电子老鼠。

两项奠基性的工作相隔十一年，都由他一个人完成：**一项让电路能做逻辑，一项让信息可以被计量。**

47

第 47 页 · 香农与信息的度量

> 克劳德·香农在国内的知名度不如图灵，但他做的两件事更直接地决定了今天技术的形态。
>
> 这两项工作分别开启了数字电路和信息论两个领域，而且相隔十一年，都由他一个人完成。

香农的两项工作

- **1937 年，硕士论文**：布尔代数（真/假、与/或/非）可以用继电器电路实现
  - 这篇论文让「用电路做逻辑运算」成为可能，是所有数字电路的理论起点
- **1948 年，《通信的数学理论》**：用 bit 度量信息量，并证明了信道容量的极限
  - 今天的手机通信、文件压缩、纠错编码，都建立在这套理论上

第 1 件补上了第 3 段留下的缺口：**开关凭什么配叫计算** —— 因为与、或、非都能用开关电路做出来。（发表时他 21 岁。）

48

第 48 页 · 香农的两项工作

第一件事在 1937 年，他的硕士论文。他指出布尔代数——也就是真与假、与或非这一套——可以用继电器电路实现：串联对应「与」，并联对应「或」。这篇论文让「用电路做逻辑运算」从设想变成工程，是所有数字电路的理论起点。写这篇论文时他 21 岁。

第二件事在 1948 年，《通信的数学理论》。他提出用 bit 来度量信息量：一件事如果只有两种等可能的结果，告诉你结果就传递了 1 个 bit 的信息。他还证明了任何信道都存在一个容量上限，低于这个上限就能几乎无差错地传输，高于它则不可能。

今天的手机通信、文件压缩、二维码的纠错，都建立在这套理论上。

哥德尔不完备定理

库尔特·哥德尔（Kurt Gödel, 1906–1978），1931 年。

哥德尔（左）与爱因斯坦（右）在普林斯顿

不完备定理：任何足够强的、逻辑自洽的形式系统，都存在在系统内既无法证明也无法证伪的命题。

49

第 49 页 · 哥德尔不完备定理

> 库尔特·哥德尔在 1931 年证明了不完备定理。
>
> 结论是：任何足够强的、逻辑自洽的形式系统里，都存在既无法在系统内证明、也无法在系统内证伪的命题。「足够强」大致是指能表达算术。
>
> 这个结果粉碎了当时数学界的一个雄心——把全部数学建立在一套完备的公理体系上，从中机械地推出所有真命题。哥德尔说明这件事做不到。
>
> 照片上和他一起散步的是爱因斯坦。两人在普林斯顿高等研究院是同事，爱因斯坦晚年说，他去研究院「主要是为了能和哥德尔一起走回家」。

第 50 页 · 三项理论的关系

> 把这三个人放在一起看，他们回答的是同一个故事的三个侧面：什么能算，是图灵；怎么度量信息，是香农；什么算不了，是哥德尔。
>
> 哥德尔的不完备定理和图灵的停机问题，在数学上其实是同一件事的不同表述——都指出形式系统内部存在不可逾越的界限。后来计算机科学里一系列「不可判定」的结论，也都属于这一族。
>
> 值得注意的是，这三项工作集中在 1931 到 1948 这不到二十年里完成，而第一台电子计算机还没有出现。理论先于机器。

第 5 段 / 共 6 段

编程语言

> 把前面确定下来的两件事摆在这儿：机器里只有 0 和 1；这串 0 和 1 本身没有意义，意义来自约定。
>
> 那么问题就很清楚了：人怎么把自己的意图，变成那一串 0 和 1。这就是编程语言干的事——它是一层约定，也是一层翻译。这一段讲这层翻译是怎么一级一级变简单的，以及这门课为什么选 Python。

机器语言与汇编语言

机器语言

```
01001000 01100101
01101100 01101100
01101111 00100000
01010111 01101111
01110010 01101100
01100100
```

汇编语言

```
section .text
  global _start
_start:
  mov edx, len
  mov ecx, msg
  int 0x80
```

这两段都在做**同一件事**：让屏幕上出现 Hello World。

52

> 接下来看语言。同一件事——让屏幕上出现 Hello World——用三种不同层次的语言写出来是什么样子。
>
> 机器语言就是一串二进制，CPU 直接执行的就是它。人几乎无法阅读，更无法维护：改一个字符就可能让整个程序失去意义。
>
> 汇编语言把这些二进制换成助记符，mov 表示搬运，int 表示触发中断。比二进制好读一些，但仍然要一条一条指挥 CPU，而且和具体的处理器绑死——换一种 CPU 就要重写。

同一件事在 Python 里的写法

```python
print("Hello world")
```

```output
Hello world
```

一行。「高级语言」的含义正在于此：编写者不必处理机器层面的细节。

> 同样的事情，Python 只需要一行。
>
> 这个落差就是「高级语言」这个词的含义：编写者不再需要处理寄存器、内存地址、中断这些机器层面的细节。
>
> 那么中间那些工作由谁完成？答案是解释器——本讲最后一段会讲到它。你写下的这一行，最终仍然会变成机器能执行的指令，只是这个转换过程不再由你来做。

语言简史

编程语言的发展趋势

- 语言是人与机器之间的翻译层
- **总体方向：更易编写，也更难写错**
- 语言之间相互借鉴 —— 一门语言里好用的特性，几年后往往出现在其他语言中
- 你现在学到的概念，十年后大多仍然适用

因此，与其纠结先学哪门语言，**不如把解决问题的方法学扎实**。

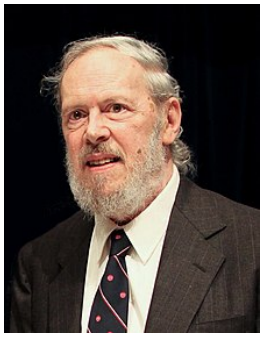
编程语言几十年来的演变有几条清晰的线索。

语言的作用是充当人和机器之间的翻译层。总体方向是让人更容易写，同时更难写错——比如自动管理内存，就消灭了一大类因为忘记释放内存导致的错误。

语言之间会相互借鉴：一门语言里被证明好用的特性，几年之后往往会出现在其他语言里。所以你在这门课学到的概念——变量、循环、函数、类、异常——十年后大概率仍然适用，只是语法细节可能不同。

这也是为什么不必纠结「先学哪门语言」。

C 语言与丹尼斯·里奇



今天几乎所有主流语言，往上追都能追到 C。

丹尼斯·里奇（Dennis Ritchie, 1941–2011）。1972–73 年在贝尔实验室做出 **C**，然后用它重写了 Unix 内核。后来斯特劳斯特卢普把类加进 C，成了 **C++**。

Python 解释器本身用 C 实现。**你调用的 `print`，其底层是 C 代码。**

> 今天几乎所有主流语言，往上追溯都能追到 C。
>
> 丹尼斯·里奇在 1972 到 1973 年间于贝尔实验室做出 C 语言，然后用它重写了 Unix 内核。在那之前操作系统通常用汇编写，换一种机器就要重写；C 让操作系统第一次具备了可移植性。后来斯特劳斯特卢普把类的概念加进 C，形成了 C++。
>
> 和这门课直接相关的是：Python 解释器本身就是用 C 实现的。你调用的 print，往下几层就是 C 代码，再往下是系统调用。
>
> 里奇 2011 年去世时几乎没有新闻报道，因为同一周乔布斯也去世了。

本课程选择 Python 的理由

- **对初学者友好** —— 学习曲线不陡，细节少（对比 C、C++、Java）
- **语法简明，功能强大** —— 同一件事，代码量常常是别的语言的三分之一
- **实用** —— 数据处理、科学计算、网站、AI，一套语法全都能干
- **开放** —— 开源，不受任何一家商业公司控制（对比 Java、C#）

以及一条现实的理由：**未来十年你很可能仍会用到它。**

这门课选 Python，有四个理由。

第一，对初学者友好。学习曲线不陡，需要一开始就掌握的细节少。对比一下：C 要求你从第一天就理解指针和内存，Java 要求你在写第一行代码前先理解类和对象。

第二，语法简明而功能强大。完成同一件事，Python 的代码量常常只有其他语言的三分之一，这在学习阶段意味着你能把注意力放在思路上而不是语法上。

第三，实用。数据处理、科学计算、网站后端、人工智能，一套语法都能用。

第四，开放。Python 是开源的，不受任何一家商业公司控制。对比 Java 归 Oracle、C# 归微软，这在长期上是有区别的。

还有一条现实的理由：未来十年你很可能仍然会用到它。

它在哪里被用

Python 的实际应用领域

- **人工智能几乎全在 Python 上** —— PyTorch、TensorFlow、JAX，主接口都是 Python
- GitHub 2025 年度报告：Python 有 **260 万贡献者**，同比增长 **48%**，在 AI 与数据科学领域领先
- **科学计算** —— NumPy 的论文 2020 年发在 **Nature**；2019 年第一张黑洞照片的成像代码（eht-imaging）就是 Python 写的
- 此外还有数据分析、网站后端、自动化运维、金融量化、生物信息

你这学期写的每一行，用的都是真实工程里在用的那门语言。

57

第 57 页 · Python 的实际应用领域

选 Python 做入门语言，有一个额外的好处：它同时也是一门在真实工程里大量使用的语言。这两件事不总是重合的——很多语言适合教学但没人在生产环境用，反过来也有。

现在它最重要的一块阵地是人工智能。PyTorch、TensorFlow、JAX 这几个主流框架，底层是 C++ 和 CUDA，但你写模型、写训练脚本用的都是 Python。这不是巧合：做研究需要改得快、试得快，而 Python 正好适合这件事。

科学计算那一条里的两个例子都值得记一下。NumPy 是几乎所有科学计算库的地基，它的论文 2020 年上了 Nature。2019 年人类拍到的第一张黑洞照片，成像那一步用的 eht-imaging 是一个 Python 包，底下是 NumPy、SciPy、matplotlib 这一套。

需要说明一点：Python 不是所有场合都最好。GitHub 2025 年的报告里，TypeScript 在总使用量上已经超过了 Python——前端和 Web 是另一片天地。Python 的强项在 AI、数据和科学计算这几块，而这几块正好是这门课之后你最可能碰到的。

最后一句更要紧：**语言只是载体。** 这门课真正教的是怎么把一个问题变成一段程序，那个能力换任何语言都还在。

Python 的来历

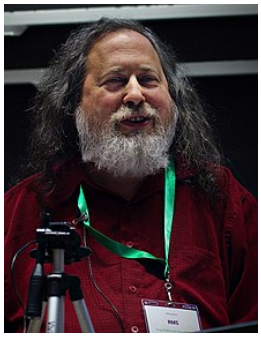
吉多·范罗苏姆（Guido van Rossum）。1989 年圣诞节假期，他为了打发时间开始写 Python。

他的车牌。名字来自 **Monty Python** 剧团，**不是蟒蛇**——虽然 logo 后来还是画成了蛇。

起因是一个假期里的业余项目。三十多年后，它成为使用最广泛的语言之一。

> Python 的来历比多数人想象的随意。
>
> 1989 年圣诞假期，荷兰程序员吉多·范罗苏姆手头没有别的事做，决定写一个自己用着舒服的脚本语言。名字取自英国喜剧团体 Monty Python，不是蟒蛇——虽然后来的官方 logo 还是画成了两条蛇。
>
> 照片里那块车牌是他本人的。一个假期里的业余项目，三十多年后成了使用最广泛的语言之一。

自由软件运动



1960 年代，硬件赚钱，软件是白送的。后来硬件利润变薄，厂商开始卖软件、并且不再给源代码。

理查德·斯托曼（Richard Stallman），自由软件运动的发起人

斯托曼的回应是：**写一套所有人都能自由使用和修改的软件**。GPL 许可证就是那时候的产物。

59

第 59 页 · 自由软件运动

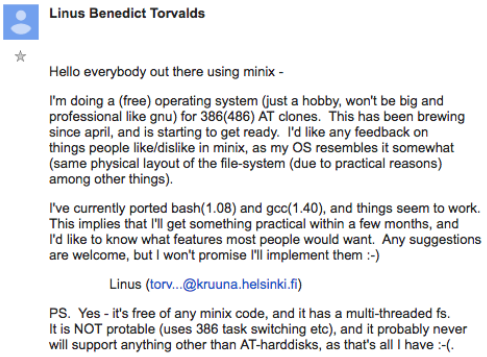
为什么这些软件不要钱，需要交代一段背景。

1960 年代，赚钱的是硬件，软件基本是附送的，源代码也随机器一起给。后来硬件利润变薄，厂商开始单独卖软件，并且不再提供源代码。

理查德·斯托曼当时在 MIT，他的回应是发起自由软件运动，目标是写出一整套所有人都能自由使用、阅读和修改的软件。GPL 许可证就是那个时期的产物：它规定，任何基于这些代码的修改版本，也必须以同样的方式开放。

注意这里的「自由」指的是使用和修改的权利，不单指价格。

Linux 的诞生



Linus Benedict Torvalds

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torv...@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

Linus Torvalds 在新闻组里宣布他在写一个操作系统。他说这「只是个爱好，不会像 GNU 那样又大又专业」。

那个「爱好」就是 **Linux**。今天世界上绝大多数服务器跑的是它。

> 这是 1991 年的一封邮件，发在一个新闻组上。
>
> 作者是芬兰的大学生林纳斯·托瓦兹。他说自己在写一个免费的操作系统，并且特意说明这「只是个爱好，不会像 GNU 那样又大又专业」。
>
> 那个「爱好」就是 Linux。今天世界上绝大多数服务器、几乎所有的超级计算机、以及每一部安卓手机，跑的都是它的内核。
>
> 这封邮件值得一看的地方在于语气：一件后来影响巨大的事情，起步时的措辞非常谦虚。

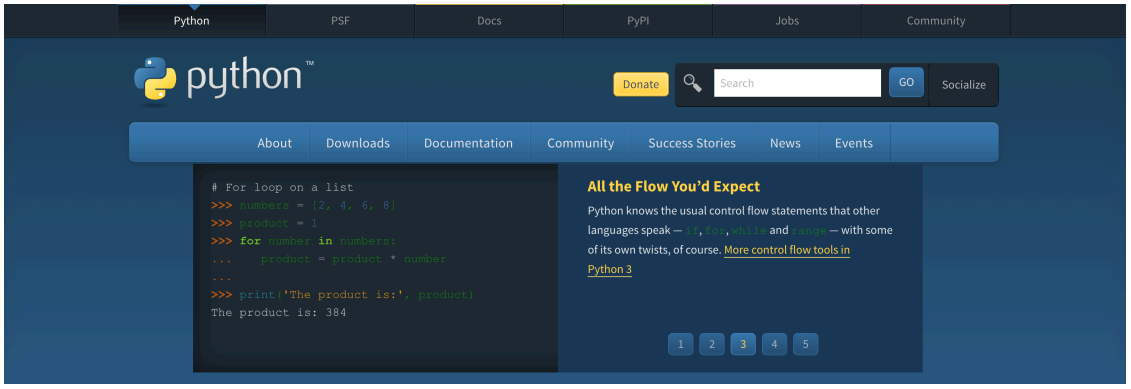
本课程使用的免费软件

- **Python 本身** —— 免费、开源
- **VS Code** —— 免费
- **Linux** —— 免费，服务器上到处都是
- **你会用到的几乎所有库** —— NumPy、pandas、matplotlib，全部免费

这些成果由几代人逐步争取而来，**今天可以直接使用**。

> 所以你今天能免费使用的这些东西——Python 本身、VS Code、Linux、以及 NumPy、pandas、matplotlib 这些库——都不是天然就免费的。
>
> 它们是几代人在具体的选择中一点点争取来的结果：有人决定公开源代码，有人设计了保证它继续开放的许可证，有人无偿维护了十几年。到你这里，直接就可以用。
>
> 这门课后面你也会写代码。开不开源、用什么许可证，将来也会是你要做的选择。

Python 官方网站



*下载走 **Downloads**。官网现在最新的是 3.14，但**这门课统一用 3.13** —— 安装包课程网站上有，Lab 1 直接下那一份。*

请认准这个域名。 搜索结果靠前的下载站常附带捆绑软件，或者给你一个被改过的安装包。

62

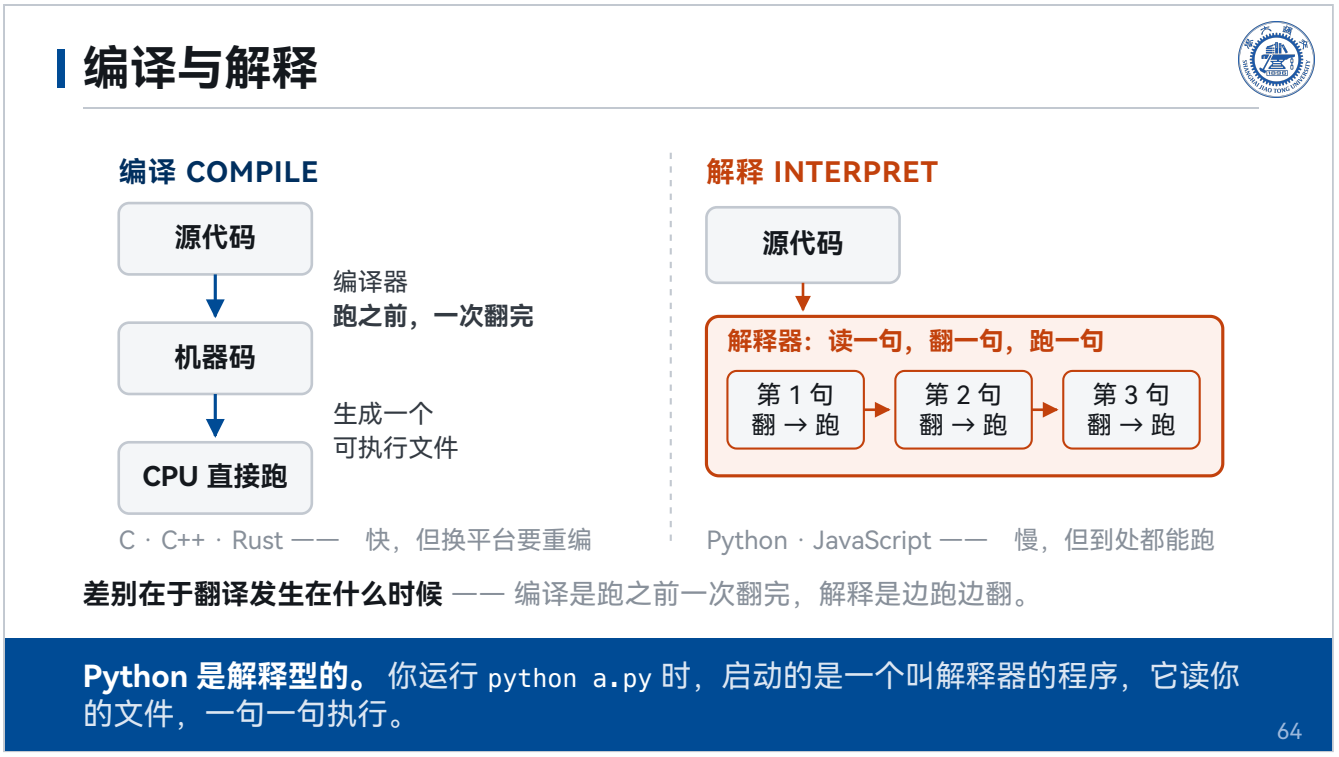
第 62 页 · Python 官方网站

> 安装 Python 请从官网 python.org 下载，认准这个域名。
>
> 搜索引擎里排在前面的下载站经常附带捆绑软件，或者提供的是被修改过的安装包。官网上同时有文档和社区入口，以后查标准库也是去那里。
>
> 有一件事先说清楚，免得你们看图的时候困惑：官网首页现在写着最新版是 3.14，但这门课统一用 3.13。原因有两个。一是全班版本一致，判题服务器跑的也是 3.13，你在本机跑通的代码交上去才不会因为版本差异出意外；二是刚发布的大版本上，第三方库往往还没跟上。
>
> 课程网站上放了 3.13 的官方安装包，还附了校验码，校园网连官网慢的时候直接下那一份。具体步骤在 Lab 1，实验课上会带着做。

第 6 段 / 共 6 段

程序的执行过程

> 回到第三段那个最关键的结论：程序和数据存在同一个地方、用同样的方式表示，所以程序本身也可以被另一个程序读取和处理。
>
> 这一段就是那句话的兑现。读你代码的那个程序有名字，叫编译器或者解释器。这一段讲它们的区别，讲「装 Python」到底装了什么，然后你写出第一个程序。



第 64 页 · 编译与解释

代码怎么变成机器上的动作，有两条路线。看图上那条虚线的两边。

左边是编译：先用编译器把整份源代码翻译成机器码，生成一个可执行文件，之后直接交给 CPU 运行。C、C++、Rust 走这条路。优点是跑得快，因为翻译在运行之前就一次做完了；缺点是换一种操作系统或处理器就要重新编译。

右边是解释：不预先翻译，由解释器读源代码，读一句、翻一句、跑一句，翻译和执行交替进行。Python、JavaScript 走这条路。优点是同一份代码到哪儿都能跑，改完立刻能看到结果；缺点是慢一些，因为每执行一次都要重新翻译。

两条路线的区别就落在一件事上：**翻译发生在什么时候。** 记住这个，后面「为什么改完代码不用重新编译」「为什么同一份 .py 复制到另一台机器就能跑」都是它的推论。

Python 属于解释型。你运行 python a.py 时，启动的是一个叫解释器的程序，由它读取并执行你的文件。

Python 环境的组成

- 装的是**解释器**（一个叫 `python` 的程序）
- 外加一套**标准库**（一堆已经写好的 `.py` 文件）
- 以及 **pip** —— 用来装别人写的库

`python a.py` 的意思是：**让解释器去读 a.py 并执行它。**

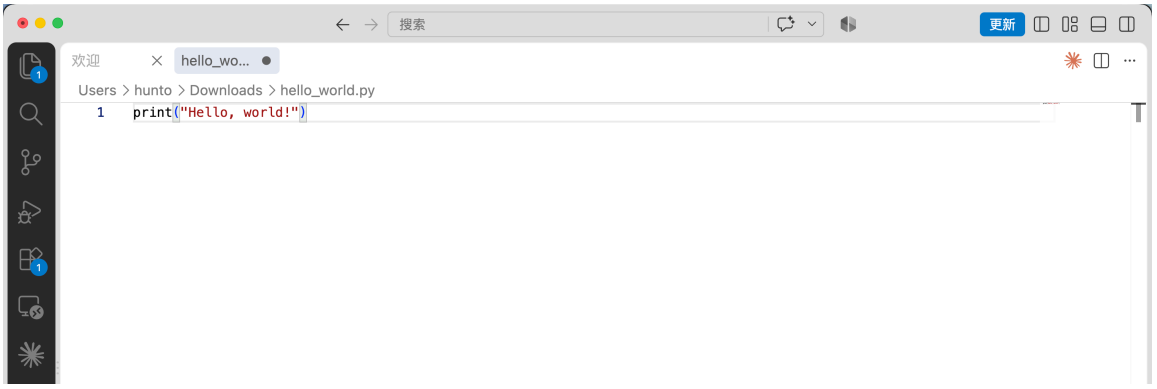
所以「安装 Python」实际安装的是三样东西。

第一是解释器，也就是一个名叫 python 的可执行程序。第二是标准库，一大批已经写好的 .py 文件，你以后 import 的 math、random、os 都在里面。第三是 pip，用来安装别人写的第三方库。

很多人一开始以为 Python 是一个有图标、可以双击打开的应用程序。不是。它是一个命令行程序，python a.py 的意思是「让解释器去读 a.py 并执行它」。把这一层理解清楚，后面讲命令行和模块时会顺利很多。

集成开发环境（IDE）

IDE（集成开发环境）= 编辑器 + 运行 + 调试 + 补全，装在一个窗口里。



*VS Code —— 这门课推荐的编辑器，免费。左边一竖条从上到下是：文件、搜索、版本控制、**运行和调试**、插件。中间是代码，一个标签页一个文件。*

第 1–7 周请关闭 AI 补全功能，安装时即可完成设置。

66

第 66 页 · 集成开发环境（IDE）

IDE 是集成开发环境的缩写，把编辑器、运行、调试、代码补全这些工具装进同一个窗口。这门课推荐 VS Code，它是免费的。

不用 IDE 也能写 Python——记事本加命令行就够——但调试功能会让你在第 4 讲之后省下大量时间。

有一件事请在安装时就做掉：关闭 AI 补全功能。VS Code 装好后通常会提示安装 Copilot 一类的插件，前七周请不要装，或者装了也把补全关掉。这不是抓违规，而是因为一旦它开着，你很难意识到自己没有在独立思考。

你的第一个程序

第一个程序

```
1  print("Hello world")
2  print("上海交通大学")
3  print("我叫 _____，学号 _____")
```

output

```
Hello world
上海交通大学
我叫 _____，学号 _____
```

`print()` 的意思是：把括号里的东西显示出来。

现在写你的第一个程序。

print 的作用是把括号里的东西显示到屏幕上。引号里的内容叫字符串，也就是一段文字；引号本身不会被打印出来，它只是用来标明「从这里到这里是一段文字」。

第二行和第三行是中文。如果它们能正常显示，说明你的环境编码没有问题；如果显示成乱码，说明终端的编码设置不对，第 12 讲会解释原因。

把第三行里的下划线换成你自己的姓名和学号，运行一次。

阅读报错信息

故意写错

```
1  pirnt("Hello world")
```

traceback

```
Traceback (most recent call last):
  File "example.py", line 1, in <module>
    pirnt("Hello world")
    ^^^^^
NameError: name 'pirnt' is not defined. Did you mean: 'print'?
```

报错已经指明：**第 1 行**，`pirnt` 未定义，并给出了拼写建议。

68

第 68 页 · 阅读报错信息

> 从第一天起就要习惯读报错。
>
> 这段代码把 print 拼成了 pirnt。运行之后 Python 给出的信息里，有三条是有用的：第一行说明是哪个文件、第几行出的问题；接下来一行把出错的那行代码原样显示出来，并用符号指出位置；最后一行是错误类型和说明——NameError 表示这个名字没有定义，并且它还给出了拼写建议 Did you mean: 'print'。
>
> 也就是说，答案基本已经写在报错里了。新手最常见的做法是看到红色文字就直接来问，跳过了阅读这一步。
>
> 读报错的顺序建议是：先看最后一行（是什么错），再看第一行（在哪里错）。

Python 之禅

试试这个

```
1  import this
```

output

点上面的「▶ 运行」看看

点一下运行 —— 它会打印出十九条 **Python 之禅**，这门语言的设计信条。

> 标准库里藏着一个存在了三十年的彩蛋：在解释器里输入 import this。
>
> 它会打印出十九条被称为「Python 之禅」的句子，是这门语言的设计信条，由 Tim Peters 在 1999 年写成。
>
> 这十九条不是语法规则，而是关于「什么样的代码算好代码」的判断。现在读可能觉得像是空话，等你写过几百行代码之后再回来看，感受会不一样。

THE ZEN OF PYTHON

Python 之禅（节选四条）

- **Beautiful is better than ugly.** 好看胜过难看
- **Simple is better than complex.** 简单胜过复杂
- **Readability counts.** 可读性是要算数的
- **If the implementation is hard to explain, it's a bad idea.**
  - 如果一个实现很难解释清楚，那它多半就是个坏主意

选出你现在最认同的一条，**学期末再回来对照**。

> 从十九条里挑四条现在就能理解的。
>
> 「Beautiful is better than ugly」和「Simple is better than complex」说的是同一个方向：能用简单办法解决的，不要用复杂办法。
>
> 「Readability counts」值得单独强调：代码写出来主要是给人读的，顺便才被机器执行。一段代码的生命周期里，被阅读的次数远多于被编写的次数。
>
> 最后一条「如果一个实现很难解释清楚，那它多半是个坏主意」给了你一个实用的自检办法：写完之后试着向别人解释你的思路，解释不清楚通常说明思路本身有问题。

第 71 页 · 随堂小测 · 第 2 轮

第二轮小测，四道题，对应今天的四个要点。我一题一题地放，每题答完当场公布答案。

第一题考摩尔定律。四个选项里有三个是同一件事的三种说法，剩下那一个不是。容易选错的是「程序运行速度每 18 个月自动快一倍」——硬件变快不等于你的程序变快，一个写得差的算法在快一倍的机器上仍然是写得差的算法。第 11 讲讲复杂度时会回到这里。

第二题问 01000001 到底是什么。如果你选了前三项中的任何一个，说明还把「表示」和「解释」当成一回事。这两者分开，是理解计算机的关键一步。

第三题考停机问题的推论。容易选错的是「死循环的程序无法被终止」——死循环当然可以终止，按 Ctrl+C 就行。停机问题说的是「事先自动判断」，不是「能不能中止」。

第四题问 python hello.py 这条命令做了什么。答对说明你理解了「解释型」：python 本身是一个程序，hello.py 是交给它的参数。选第一项的人是把 Python 当成了编译型语言。

回答开场

所以，计算机凭什么能算

- **因为「算」可以拆成一串只有通断两种结果的判断** —— 这是香农 1937 年证明的：与、或、非都能用开关电路做出来
- **因为这样的机器不需要为每个任务重造** —— 程序和数据一样是 0 和 1，换任务只是换一段存进去的 0 和 1
- **因为「能算什么」是有定论的** —— 图灵证明了任何机械可算的东西，那条纸带都算得出来；同时也划出了算不了的那部分
- **而它之所以快到有用**，是因为开关越做越小 —— 摩尔定律那条线

这四条里没有一条讲到 Python。**语言是后面的事，今天这一讲讲的是它底下那一层。**

第 72 页 · 所以，计算机凭什么能算

今天开头问的是：1943 年招的 computer 是人，今天这台机器凭什么能算。现在正面回答。

第一，因为「算」这件事可以拆成一串只有通断两种结果的判断。这不是比喻，是香农 1937 年那篇硕士论文证明的：与、或、非这套逻辑，都能用继电器电路实现。有了这一条，开关才配叫计算。

第二，因为这样的机器不需要为每个任务重造。程序和数据一样是 0 和 1，存在同一个地方，换任务只是换一段存进去的 0 和 1——这是冯诺依曼那份报告里最关键的一句。ENIAC 换任务要重新插线，有时候要好几天。

第三，因为「能算什么」是有定论的，不是碰运气。图灵证明了：任何能机械地算出来的东西，那条纸带都算得出来。同一套理论也划出了算不了的那部分，比如停机问题。

第四，它之所以快到有用，是因为开关越做越小，也就是摩尔定律那条线。注意这一条是**工程**，前三条是**原理**——原理决定了能不能算，工程决定了算得快不快。

你会发现这四条里没有一条讲到 Python。语言是后面的事。今天这一讲讲的是它底下那一层，而这一层十六周都不会变。

其他三条经验法则

- **梅特卡夫定律** —— 网络的价值与节点数的平方成正比
  - 这解释了为什么社交平台赢家通吃
- **墨菲定律** —— 如果一件事有可能出错，它就一定会出错
  - 写程序的人对这条会有越来越深的体会
- **Knuth 定律** —— 过早的优化是万恶之源
  - 这条我们在第 11 讲会认真讲

第 73 页 · 其他三条经验法则

> 顺便介绍三条流传较广的经验法则，它们不是定律，但常被引用。
>
> 梅特卡夫定律说，一个网络的价值与节点数的平方成正比。它解释了为什么社交平台容易赢家通吃：用户越多，对新用户的吸引力越大。
>
> 墨菲定律说，如果一件事有可能出错，它就一定会出错。写程序的人对这条体会最深——你以为不可能为空的输入，一定会有人传空值进来。
>
> Knuth 定律说，过早的优化是万恶之源。意思是在没有测量之前就去优化代码，通常既浪费时间又让代码变难懂。这条在第 11 讲会认真展开。

回顾

本讲小结：一条线

- 人自己算 —— **算不动，而且会算错**（所以才要造机器）
- 机器靠**开关**接手 —— 两百年只在把开关做小做快
- 开关只有通断 —— **所以机器里只有 0 和 1**
- 0 和 1 本身没有意义 —— **意义来自约定**
- **程序也是 0 和 1**，和数据存在一起 —— 所以程序能被程序读，才有了解释器
- 至于**能算什么**，图灵在机器造出来之前就划好了边界

第 74 页 · 本讲小结：一条线

> 今天这一讲请顺着这条线走一遍，它就是今天全部内容的骨架。
>
> 人自己算，算不动而且会算错——所以才要造机器。机器靠开关接手，两百年的技术进步全花在把开关做得更小更快上，开关本身的性质没变。
>
> 开关只有通和断，所以机器里只有 0 和 1。而这串 0 和 1 本身没有意义，意义来自约定——这是今天最要紧的一句，第 2 讲会展开。
>
> 程序也是 0 和 1，而且和数据存在同一个地方，所以程序能被另一个程序读取和处理，才有了编译器和解释器。你装的 Python 就是那个解释器。
>
> 最后一条是边界：能算什么、不能算什么，图灵在第一台电子计算机造出来之前就划好了。
>
> 这六步能连着说下来，今天就没白上。

LAB 1

本次实验：环境搭建与第一个程序

- **装 Python 3.13 和 VS Code**，跑通 `print("Hello world")`
  - 两种方式各跑一次：VS Code 的 ▶ 按钮，和终端里 `python hello.py`
- **A 段 · 必做** —— 自我介绍、学号转进制、猜四个算式、故意制造三种报错
- **B 段 · 深入** —— 十行代码看混沌：初值差百万分之一，二十五轮后面目全非
- **C 段 · 基本功** —— 命令行 `pwd ls cd`。每周一小段，**期末要考**
- **走通一次提交** —— 用学号注册评测平台（不要口令），交「小作业 1」

三段**同时开放**，不必按顺序做完；以后每周都是这个结构。**装环境会吃掉大半时间，所以本周三段都轻。**

75

第 75 页 · 本次实验：环境搭建与第一个程序

本周实验的正题是把环境装好，但结构你要先听清楚——以后每周都是这个结构。

第一件事是装 Python 3.13 和 VS Code，跑通 print("Hello world")。注意要用两种方式各跑一次：VS Code 右上角那个三角形按钮，和终端里敲 python hello.py。它们干的是同一件事，运行按钮只是替你敲了那行命令——今天讲解释器那一段说的就是这个。

然后题目分三段，**同时开放**，不必按顺序做完。A 段必做，覆盖本周知识点，平台自动判：自我介绍、把学号转成各种进制、四个算式先猜再跑、以及故意制造三种报错（拼错 print、用中文括号、引号只写一半），把报错的第一行抄下来。

B 段是一道大题，分几问递进。这周这道很有意思：十行代码，同一个式子迭代二十五轮，初值差百万分之一，最后面目全非——这就是蝴蝶效应。不用现在懂，跑出来看就行。

C 段是计算机基本功，命令行、版本控制、远程登录这些，每周十五分钟。这周是四个命令：pwd、ls、cd、cd 两点。**这一段期末要考**，别跳过。

最后一件事最要紧：**走通一次提交**。用你的学号到评测平台注册，不需要口令，学号在选课名单里就能注册；然后进「小作业 1」。以后每周的东西都交到对应的小作业里。小作业 1 覆盖前三周，10 月 10 日截止——但别拖，这周的东西这周做完。

装环境会吃掉今天大半时间，所以本周三段都轻。遇到问题请在实验课上找助教，不必自己在宿舍折腾到半夜。

下一讲预告

计算机只有 0 和 1，那它怎么表示 -3、3.14 和「你好」？

下一讲：字符与数。你会看到 0.1 + 0.2 != 0.3，并且明白这不是 Python 的 bug。

> 下一讲讲字符与数。
>
> 开头的问题是：计算机里只有 0 和 1，那它怎么表示 -3、3.14 和「你好」？ 负号、小数点和汉字都不是 0 和 1，它们必须被编码成某种约定。
>
> 你还会看到 0.1 + 0.2 不等于 0.3，并且会明白这不是 Python 的缺陷，而是二进制表示小数时的必然结果。