前言

第一部分，不需要写代码，只要能够运行即可

双击安装Noto.tff字体，文章中全部使用Noto字体

本教程无需前置知识，内容繁杂但都很基础。每一章内容都可以进行更深入和广泛的研究。
本文中的Processing和Arduino开发都使用Python，主要原因是学生后期会学习AI知识。
Processing原生语言为Java，Arduino原生语言为C++

注意

1. **本文中的所有代码都是在Windows上生成的**
2. **理论上可以直接在Mac或者Linux、树莓派等设备上使用**
3. **本文使用Google-Noto-Sans作为跨平台字体**
4. **由于是草稿，后期还会大幅修改，因此各章节未编号，未添加图片，未完整排版**

第一部分 Processing 概述

第一章 Processing 开发环境准备

本章内容

1. 安装Java、Python、Py5、VS Code、Jupyter Notebook
2. 学习使用Jupyter Notebook交互式学习环境
3. 学习使用VS Code开发环境（IDE）
4. 运行第一个Python程序
5. 运行第一个Py5程序

1. Processing介绍

你是不是也曾经想过：如果你现在已经会写代码，真的还有必要学 Processing 吗？“我已经掌握了 Python、Java、甚至 C++，还需要动手学‘这个给艺术家和设计师准备的玩具’？”但你有没有注意到，我们很多编程时的“酷想法”，在传统开发环境下往往会被实现细节拖慢？你是否遇到过，仅仅是想把数据变成生动的动态图像，或者快速做个算法原型，就要纠结窗口管理、底层绘图库、兼容性适配问题？Processing 的存在正好能让你跳出繁杂的限制，让有创意的想法用极少的代码、最直接的方式变成可以“眼见为实”的作品——不想感受一下“写代码像画速写”的快感吗？

如果你是一位传统艺术家，又有没有质疑过——“我从来没写过程序，数字创作离我会不会太远？”其实，Processing 的设计初心就是让艺术家和设计师不用考虑晦涩难懂的计算机底层细节，也能轻松用“代码”这支画笔，把静态的创意变成会动、能互动、可以被观众参与的数字作品。你有没有幻想过，让你的作品“动态起来”，甚至能和观众实时互动、在线传播，真正打破画布和舞台的边界？Processing，就是为你准备的数字艺术“万能工具箱”。

如果你已经是交互艺术领域的创作人，你肯定也在思考：“怎样才能用更短的时间、学更少的复杂技能，把我的想法又快又稳地做出来，和各种硬件设备、传感器打交道，还能充分发挥我的视觉美学？”Processing 不仅允许你轻松捕捉观众每一个动作、声音或触碰，甚至能和 Arduino 等硬件打通，实现软硬一体的交互现场。至于投影mapping、传感器互动、甚至实时数据可视化，这些曾经需要大团队和复杂工程实现的东西，现在你一个人“十分钟就能玩起来”。难道你不想自己试试吗？

在现代数字艺术与交互装置设计领域，Processing 已经成为极具影响力且不可或缺的核心工具。作为一个开源、免费的图形库和集成开发环境（IDE），Processing 最初源于麻省理工学院媒体实验室，由 Casey Reas 和 Ben Fry 于 2001 年共同开发，其目标即是为电子艺术、新媒体艺术和视觉设计等领域的创作者提供一款易用的编程工具。它通过极大程度地简化编程语法和接口，让完全没有计算机科学背景的设计师、艺术家与教育者也能轻松入门数字创作，在短时间内完成视觉效果、动画、交互装置等创新实践。与诞生之初动辄必须掌握 C++ 或 Java 等复杂底层技术的时代相比，Processing 采用了“草图”模式并集成了许多经过高度抽象的功能如绘图、数据处理等，从而显著降低了数字艺术创作的门槛，使非专业编程人员亦可投身数字媒介艺术之中。2004 年以来，随着其社区的开放与全球化，以及 Processing 基金会的设立，Processing 不断迭代和壮大，推动了 p5.js、Wiring 等多平台项目的兴起，实现了面向硬件、网络、物联网、实时表演等多样应用场景的跨领域创新。而在技术层面，Processing 兼具简化且强大的开发语言特性，允许用户以最少的代码实现丰富的视觉表现和动态交互，同时兼容主流操作系统，极易与 Arduino 等开源硬件集成，为新媒体艺术与交互设计开拓了新的疆界。

Processing 为什么一开始要基于 Java？这其实是团队经过深思熟虑后的决定。在当时，Java 的最大优势就是跨平台、安全、生态庞大？Processing 利用 Java “写一次到处跑”的特性，让你无论用什么系统都可以无障碍创作，而且继承 Java 强大的图形和网络能力。而且，Processing 也屏蔽了 Java 冗杂难学的部分，通过简约的语法和 API，降低了初学者的上手门槛。是不是既兼顾技术强大、又不用“被代码绑架”？

那为什么本书又特别强调用 Python？你或许好奇，“是不是多此一举，或者只是图个新鲜？”其实恰恰相反——Python 的简单易学几乎是公认的，无论你有没有编程基础，入门都无压力。再者，无论是 Processing 还是 Arduino，如今都为 Python 提供了官方支持，这意味着你不仅能做可视化交互，还能一步到位连接现实世界的各种智能硬件。更关键的是，如果你以后想进阶人工智能、机器学习领域，Python 现在已经成了最主流的开发语言。也就是说，学会了 Processing 和 Python，不仅满足了创意表达，未来无论走工程、艺术还是 AI，你都有了足够多的选择和竞争力。还不觉得这是一条特别划算的学习路径吗？

通过本书学习，你会发现自己不仅能用 Python 模式写出属于自己的视觉交互作品，还能洞悉游戏编程的基本原理，亲手实践到底什么是交互装置、物联网，以及这些技术如何和日常生活乃至前沿艺术深度融合。你是不是已经在想，未来自己的第一个作品会是什么样？现在，就让我们一起在代码与创意之间，开启一段看得见成果、玩得了技术、讲得出故事的数字创作旅程吧！

2. 安装基本的运行环境

2.1 安装JAVA运行时

JAVA是一种面向对象的高级编程语言和计算平台，由Sun Microsystems公司于1995年首次发布。它最大的特点之一是“跨平台性”，即所谓的“Write Once, Run Anywhere”（一次编写，到处运行）。这种跨平台特性源于JAVA程序会被编译成字节码（bytecode），运行时由JAVA虚拟机（JVM）执行，因此无论在Windows、macOS还是Linux等不同操作系统，只要安装了JVM，JAVA程序都能顺利运行。

Processing是一款基于JAVA语言开发的开源可视化编程环境，广泛应用于视觉艺术、图形设计与编程教育等领域。由于Processing本身是基于JAVA开发的，并且所有Processing编写的项目最终也会在JAVA环境中执行，因此在使用Processing之前，必须要确保计算机已经正确安装了JAVA运行环境，也就是通常所说的JAVA运行时（Runtime Environment, 简称JRE）或JAVA开发工具包（JDK）。

- 名词汇总
  - **Java虚拟机（JVM）：**
    JVM作为Java程序运行的核心，将JAVA代码转换为特定平台上的机器码。
  - **Java运行时环境（JRE）：**
    JRE包含了JVM及Java核心类库，对于仅需运行JAVA程序的用户，安装JRE即可。
  - **Java开发工具包（JDK）：** 对于开发者来说，JDK不仅提供了运行时所需的JRE，还附带了编译器及调试工具等开发工具

开发工具。

推荐从JAVA的官方网站（Oracle官网）下载最新版本的JAVA。本文使用的Processing（Python，Py5）要求至少安装JAVA 17版本，而本书撰写时JAVA的最新版本为JAVA 24。下载对应平台的安装包后，双击运行安装程序，根据提示完成安装即可。对于不同系统，JAVA的安装目录会有所不同，记下该路径以便后续配置环境变量。配置环境变量可确保系统及开发工具（如IDEA、Eclipse、 VS Code或Processing）能正确调用Java工具。

- Windows具体步骤如下：

  1. **打开系统属性：** 在“此电脑”或“我的电脑”图标上右击，选择“属性”，进入后选择“高级系统设置”，然后点击“环境变量”。

  2. **新建JAVA_HOME变量：**
     在系统变量中点击“新建”，变量名填写为 `JAVA_HOME` ，变量值填写为JDK安装目录（例如：D:\Program Files\Java\jdk-24）。

  3. **打开命令提示符：**
     使用快捷键Win+R，输入“cmd”，打开命令提示符窗口。

  4. **检查Java版本：**
     在命令行中输入命令：
     ```
     java -version
     ```
     若安装正确，将显示类似以下信息：
     ```
     java version "24.0.1" 2025-04-15
     java(TM) SE Runtime Environment (build 24.0.1+9-30)
     java HotSpot(TM) 64-Bit Server VM (build 24.0.1+9-30, mixed mode, sharing)
     ```

  这种输出信息证明JAVA运行环境配置成功413。

Mac、Linux和树莓派用户请自行检索，并设定JAVA_HOME值。提示：首先确认本机使用的终端类型，才能选择合适的指令进行配置。

2.2 安装Processing（Python，Py5）开发环境

Python作为一种高级编程语言，以其简洁的语法和庞大的库生态系统，已成为数据科学、人工智能（AI）和快速原型开发领域中的首选语言。同时，Processing历来以其强大的实时图形渲染能力和交互式设计而闻名，尤其在视觉艺术和创意编程方面占据重要地位。随着科技的发展，尤其是在人工智能和数据科学领域的快速进步，Python已经超越JAVA，成为目前全球范围内最受欢迎的编程语言之一。考虑到未来学习和技术应用的广阔前景，本书决定采用Python作为主要开发语言，以便读者能够更好地掌握现代编程工具并具备更高的竞争力。

Processing项目很早就支持了Python语言。在官方的Processing IDE中，用户可以直接切换到Python模式。此外，也有像Py5这样独立与Processing IDE的Python库，完美复刻和扩展了Processing的功能，不仅可以用于创作可视化交互项目，还能够在任何Python程序和各种开发环境中灵活集成，而无需运行Processing IDE。这使得Python与Processing的结合变得更为高效和强大。

Py5是一个开源Python库，它的出现不仅突破了传统Processing依赖于特定IDE的局限，还将Processing的创意编程能力与Python灵活的开发生态系统相结合。

- 名词汇总

  - **Python：**
    一种广泛流行的全新高级编程语言。

  - **Python库：**
    由他人（或自己）编写的、可以直接拿来用的Python功能模块，能够极大提升开发效率。
    可以简单理解为Python的功能扩展“插件”。

  - **Py5：**
    一个专门为Python用户设计的Processing功能库，使得用户可以用纯Python代码实现图形和交互编程。

- 具体安装步骤如下：

  1. 安装Python
     要在本地电脑上使用Python开发Processing项目，首先需要下载安装Python。可以访问Python官方网站下载适用于你操作系统的Python安装包。Processing要求Python的最低版本为3.9，而本书编写时的最新稳定版本为Python 3.13。下载后双击安装，一路选择默认选项即可，在安装过程中，**务必勾选"Add Python to PATH"**。

  2. 验证Python安装
     在Windows平台上，安装结束后打开命令提示符（CMD），输入 `python -version` 指令，如果系统显示出类似 `Python 3.13.0` 这样的版本号输出，即说明Python已经成功安装。

3. 安装Py5库
   接下来，安装Py5库，可以直接使用Python自带的包管理工具pip。在命令提示符中输入：`pip install py5`，等待命令执行完成，pip会自动下载并安装好Py5以及相关依赖库。

4. 验证Py5库安装
   安装结束后，可以**再次打开命令提示符**输入`python`进入Python的交互式编辑模式（提示符变为`>>>`），然后输入：`import py5`，
   如果直接回车后没有出现任何错误提示（如ModuleNotFoundError等），说明Py5安装成功。

至此，Processing（Python, Py5）开发环境就安装结束了。

不过，好的作家也不会只用一支笔写完一本书。现代科技为作家们提供了各种工具，比如电子笔、文字识别软件、资料管理工具等，这些都极大提升了创作效率。同样地，对于程序开发人员来说，仅靠最基础的代码编辑工具早已无法满足日益复杂的项目需求，这时就需要引入更强大、更贴合实际开发流程的工具——IDE。

IDE，全称为“集成开发环境（Integrated Development Environment）”，是专为编程设计的软件工作平台，集代码编辑、自动补全、项目管理、调试、运行、版本控制等多种功能于一体。它能够帮助开发者高效编写、管理和调试代码，大幅提升学习和开发的体验。

Processing 官方虽然自带了一个轻量级的默认IDE，适合初学者快速上手和进行简单实验，但其功能相对有限。考虑到很多读者未来可能会涉及更复杂的项目，或者希望进一步进入如科学计算、数据分析、Web开发等更广泛、主流的技术栈，掌握主流且专业的IDE将极具优势。因此，本文选择介绍目前业界应用广泛、兼容性良好的 Jupyter Notebook 和 Visual Studio Code（VS Code）作为开发环境。

Jupyter Notebook 是一款交互式的编程笔记本，非常适合数据分析、科学计算、教学和探索性编程。它可以在网页界面中编写、运行与可视化代码，并便于文档化、讲解和分享。

VS Code 则是一款轻量但功能强大的现代编辑器，支持众多编程语言、可高度扩展、跨平台、内置Git版本控制，适合专业开发、多人协作及大型项目管理。

两者都拥有庞大的用户社区和丰富的插件生态，能极大提高学习和开发效率。

3. 使用Jupyter Notebook进行交互式程序设计学习

在传统的程序开发教学中，教师通常会将完整的代码打包发给学生，要求大家直接运行和学习。这种做法虽然保证了代码的完整性，但也有很大局限性：一方面，如果只给代码片段，学生往往无法理解整体逻辑和运行过程；另一方面，完整项目往往体积庞大、结构复杂，初学者很容易感到迷茫和畏惧，难以快速抓住重点、循序渐进地学习。

Jupyter Notebook 的出现很好地解决了这一难题。Jupyter Notebook 是一种基于网页的交互式编程环境，最初由 IPython 项目组发起和开发。Jupyter Notebook 的最大特点在于“交互式细粒度编程”：代码和文档可以按单元格（Cell）逐步组织和运行，用户不仅可以随时运行某个片段，还能即时看到输出、调试结果和可视化图表。这样一来，编程学习过程便可以完全模块化、结构化，阅读和操作更加直观灵活，极大提升了学习和教学效率。

Jupyter Notebook 的文件以 `.ipynb` 作为后缀。其内容通常由两类单元格组成：**Markdown Cell** 和 **Code Cell**。

- Markdown Cell 用于书写文字说明、公式、图片、标题及注释，支持 Markdown 语法，方便教师或作者进行详细讲解。
- Code Cell 则专门用于编写和运行 Python 代码，每个代码单元格都能独立执行并展示输出。这样的设计使得讲解与实验紧密结合，理论和实践无缝衔接。

本书默认使用 Jupyter Notebook 进行写作与演示。不过，每个 Code Cell 内部的 Python 代码都是完整且独立可运行的，没有前后文依赖。

如果你正在阅读本书的 PDF 版本，只需将对应的 Python 代码复制到任意 Python IDE 里，按照书中的分步讲解逐个粘贴运行，即可获得相同的学习体验和结果。

- 名词汇总
  - **Jupyter Notebook：**
    一种网页端交互式编程环境，支持代码、文本和可视化内容混排。
  - **Markdown Cell：**
    Jupyter Notebook 中用于书写文本、标题、公式和注释的单元格。
  - **Code Cell：**
    Jupyter Notebook 中用于编写与运行代码的单元格，输出结果直接显示下方。
- 具体安装步骤如下：
  1. 首先，确保电脑中已经正确安装了 Python 环境（推荐 3.9 及以上版本）。
  2. 打开命令提示符（Windows）或终端（Mac/Linux），输入以下命令安装 Jupyter Notebook：`pip install notebook`
  3. 安装完成后，在命令行输入：`jupyter notebook`。浏览器会自动打开一个新的窗口，显示 Jupyter Notebook 主界面。如果能看到管理页面并新建、编辑 `.ipynb` 文件，说明 Jupyter Notebook 已经安装成功并可以正常使用。

现在，你就可以开始使用 Jupyter Notebook 进行更高效、更灵活的 Python 学习与开发了。

4. 使用VS Code进行多语言跨平台开发

为了完成本书的学习，你需要先简单了解市面上的几款主流IDE：

- Processing IDE ：Processing的官方自带编辑器，界面极简，上手快，适合初学者可视化和交互实验，但功能略显单薄，项目和插件扩展能力有限。
- PyCharm：专业Python开发利器，专为Python打造，带有完整的项目管理、调试、测试、自动补全和环境配置功能，适合中大型Python项目，但体积偏大，对新手可能有点“压顶”。
- Eclipse：JAVA及其他主流语言的老牌IDE，开源免费、功能强大，插件体系庞大，适合管理大型复杂项目，但上手曲线略陡、UI略显“复古”。
- Arduino IDE：专为Arduino单片机开发设计的编辑器，界面友好，上手简单，适合硬件初学者，但与主流桌面开发环境割裂，日常多领域开发略显捉襟见肘。

看着这些“各自为政”的IDE，天哪，你是不是已经在内心默默许愿：“求求了，让我只用一个软件吧！”

想象这样一个令人头大的画面：你一边要用Processing IDE写交互效果，一边要跑Python处理点AI数据，隔壁同学又喊你帮忙用Arduino IDE制作一个交互装置，结果你的电脑桌面上堆满了各种IDE和窗口。左边是Processing IDE（Processing），右边是PyCharm（Python），一会还得分心切到Eclipse（JAVA），甚至Arduino IDE（Arduino）也来掺一脚。此情此景，天哪，你是不是已经开始怀疑人生，搞不清到底该往哪个窗口里敲代码？

如果有一种工具能把所有开发环境揉到一起，不就世界大同了吗？好消息：VS Code就是你的超级救星！

VS Code（Visual Studio Code）是一款由微软推出的现代、开源、免费的编辑器。它以轻量级著称，启动秒开不拖沓，却又“身轻体壮”——通过丰富的扩展（Extension）系统，几乎可以支持市面上你想得到的所有主流编程语言、框架与工具。更棒的是，VS Code 跨平台支持 Windows、macOS 与 Linux，凭借高扩展性、极速响应、优雅界面与巨大社区，被全球开发者誉为“开发界的瑞士军刀”，一跃成为编程必备工具。

默认情况下，VS Code其实只是一个“骨感”的代码编辑器，但通过各种扩展插件（Extension），它可以化身为超级 IDE，支持调试、文档、AI 助手、远程开发等各种功能。无论你要搞 Processing（JAVA）、传统 JAVA 工程、Python、Py5 还是 Arduino 单片机开发，都可以通过安装相应扩展，一键集成到 VS Code 里！这样，无论课堂学习还是实际项目，只需要专注一个软件即可，大大降低了环境切换带来的混乱，节省了宝贵的摸鱼时间，学习和开发一次到位。

- VS Code 安装与必备扩展

1. **下载安装 VS Code**
   访问 VS Code 官网，选择对应系统下载安装包，双击按照向导安装即可。
2. **（可选）登录并启用 Copilot AI 辅助开发**
   启动 VS Code 后，点击左侧“扩展”图标，搜索并安装 GitHub Copilot。安装后会引导你用 GitHub 账号登录，开启智能自动补全与AI代码助手。
3. **安装 IntelliCode 扩展**
   依然在扩展市场搜索 IntelliCode，安装后，享受微软自家 AI 辅助的智能补全和代码建议，提升开发效率。
4. **安装 Python 扩展**
   在扩展市场搜索 Python，选择官方 Microsoft 出品的 Python 扩展，安装后可高亮语法、跳转定义、调试、虚拟环境管理。
5. **安装 Jupyter 扩展**
   在扩展市场搜索 Jupyter 并安装后，可直接在 VS Code 里新建和编辑 .ipynb 文件，实现交互编程与可视化。

安装完以上扩展后，就可以用 VS Code 一站式管理和开发 Processing（JAVA模式）、Python、Py5、Arduino 甚至未来更多项目类型，统一开发体验，从此告别“窗口切换地狱”！

5. 第一个Python程序

目前为止，如果一切顺利的话，你应当完成了以下内容：

- 安装JAVA运行时，从而运行Processing应用程序
- 安装Python和Py5，使你能够用Python进行Processing开发
- 安装Jupyter Notebook，以便阅读本文档
- 安装VS Code，帮你高效的敲代码

再一次，如果一切顺利话，我们现在可以开始你的第一个Python程序，我将通过这个程序向你简单介绍VS Code的使用和Python应用程序的基本代码结构。

如果实在自行无法解决安装环境的问题，可以附带截屏发送到我的邮箱、Github、或者小红书私信~

启动 VS Code 后，你会看到整洁而现代的主界面。让我们先来简单介绍各个区域的功能：

界面区域	位置	主要功能
编辑器	中央	是你编写和浏览代码的主要窗口，可以同时打开多个文件标签页
主侧边栏	左侧竖排图标	文件管理、搜索、版本控制、扩展（插件）、运行与调试视图等，点击图标可以快速切换各类面板
次侧边栏	右侧扩展视图	当前为Copilot AI Chat交互界面
状态栏	下方横条	显示当前文件信息（如行号、编码格式、Python环境）、Git分支、错误提示等实用状态，实时反馈工作状态和系统信息
面板	下方扩展视图	包括“终端Terminal”（运行命令和代码）、“输出Output”、“调试Console”、“问题Problems”等，当你运行代码或调试程序时，都可以在这里看到相应的输出结果或错误

以上这些面板都可以在 VS Code 右上方进行快捷打开或关闭，也可以在 View（视图）菜单中打开或关闭。

创建并运行第一个 Python 文件

1. 在左侧“资源管理器”点击“新建文件”图标，或用快捷键 **Ctrl+N** 新建一个文件，然后通过菜单“文件-另存为”将其保存为 `example.py` 。
2. 在编辑器中粘贴如下代码，并随时保存（**Ctrl+S**）：

```
print("hello world") # 这是一个简单的 Python 程序将打印 "hello world" 到控制台
```

3. 在代码文件的右上角，你会看到一个绿色的“运行”按钮（“▶”图标）。
4. 程序运行后，在 **下方的“终端”面板** 中就能看到输出结果。

或者你也可以直接在Jupyter Notebook，也就是在本文档中运行这段代码

1. 使用VS Code打开本文件
2. 点击右上角的Jupyter Notebook Kernel选择按钮
3. 选择 `python environment - python 3.13` ，也就是你之前安装的Python
4. 成功启动Kernel的话，会在右上角看到Python版本号
5. 选中下面的 Code Cell，在 Cell 左侧外面一点会出现“运行”按钮（“▶”图标）。

In [1]:
print("hello world") # *这是一个简单的 Python 程序将打印 "hello world" 到控制台*
hello world
In [2]:

```
# 这是一个使用 Py5 库绘制矩形的 Processing 程序
import py5  # 引入第三方库，Py5，使用Py5包装好的Processing功能

def setup(): # def 表示设计了一个函数（功能模块），setup()是Processing的一个标准函数
    py5.size(400, 400) # 设置画布大小为400x400像素
    py5.background(255) # 设置背景颜色为白色
    py5.stroke(0) # 设置描边颜色为黑色
    py5.stroke_weight(2) # 设置描边宽度为2像素
    py5.fill(100, 150, 250) # 设置填充颜色为RGB(100, 150, 250)
    py5.rect(50, 50, 300, 300) # 绘制一个矩形，位置在(50, 50)，宽度和高度均为300像素

def draw():
    '''
    draw()函数在setup()之后循环执行，通常用于绘制动画或动态效果
    在这个例子中，draw()函数为空，因为我们只需要绘制一次静态矩形
    '''
    pass # pass语句表示什么都不做，留空函数体

py5.run_sketch() # 函数调用，运行Py5程序，开始执行setup()和draw()函数
```

初识Python注释与程序结构

你需要掌握的第一个Python知识点就是“注释”与代码结构：

- **单行注释**：用 `#`，其后内容不会被程序运行，用于解释说明每行代码的作用。
- **多行注释/文档字符串**：用三个单引号 `'''` 或三个双引号 `"""` 包裹，常用于解释函数、类、模块的功能（即“文档字符串”）。

合理的使用注释，不仅能够帮助别人看懂你写的代码，更重要的帮助你自己理解多年前写的**乱麻**

可以看到，上面的Python程序大体有三大部分构成：

1. **import** 语句：导入模块或库，如果不使用第三方库则没有这个部分。
2. **由 `def` 开始的函数定义**：函数就像“功能块”或“工具”，比如“setup”就封装了初始化Processing的功能；每个函数实现一个相对独立的功能。
   如果你自己不打算定义新的功能，仅仅是调用别人写好的函数，比如 `print`，那么也没有这个部分。
3. **函数调用**：现在有了许多厨具（函数）和食材（数据），你可以像大厨一样“操刀上阵”，用这些工具和食材组合成一道成品。 这就是函数调用，比如前面你调用的 `print` 函数，将数据 `hello world` 传递给它，从而在屏幕上显示这行文字。

在实际的开发过程中，一定会出现各种代码错误，比如字母写错了，数据输入错误，或者更严重的逻辑错误，所有的开发环境都提供的错误反馈机制，帮助开发者解决这些问题。

制造并理解一个错误信息

运行下面这段代码。

```
In [ ]:
import pY5 # 这里的pY5是错误的，应该是py5，Python区分大小写

def setup():
    py5.size(400, 400)

def draw():
    pass

py5.run_sktech() # 这里的函数名拼写错误，应该是run_sketch()
```

会发现下方终端面板里出现了红色的**错误提示（error）**：

```
ModuleNotFoundError: No module named 'pY5'
```

以及

```
AttributeError: py5 has no field or function named "run_sktech". Did you mean "run_sketch"?
```

这就是**错误调试**的过程。错误信息通常包括：

- **错误类型**（如 NameError、SyntaxError、TypeError 等），说明出了什么问题；
- **行号**：通常提示错误出现在代码的哪一行；
- **错误描述**：如这里的 `No module named 'pY5'`，意思是“没有名为pY5的库”。

这段错误信息就是“程序为你敲响的警钟”。通过分析这些提示，你可以快速定位代码问题，例如此例就是“函数名拼写错误”，只需改正即可。学会解读并利用错误信息，是每个程序员走向高手的必经之路——它能帮助你快速发现、修正问题，写出更稳健的代码。

本章总结

本章知识点汇总

1. Jupyter Notebook 和 VS Code 在 Processing 开发中的作用

- **Jupyter Notebook**：就像一本可以边写边运行的小笔记本，特别适合写点短代码、做实验、边试边看结果，很适合上课演示或者自己学点新东西。
- **VS Code**：这是个超级好用的多编程语言大本营。

2. Python 注释的用法和作用

- **怎么写**：单行注释在代码前面加个 `#`，比如 `# 这是注释`；多行注释一般用三个单引号 `'''` 或双引号 `"""` 包起来。
- **主要作用**：给自己或者别人解释代码，说明"这段在干啥"；写给人看的！

3. 什么是函数？

- **一句话概括**：函数就是一段能反复用的小工具，有点像厨房里的微波炉，把东西丢进去自动帮你搞定一件事情。
- **具体点说**：它接受输入（参数），做点事，然后给你一个结果（返回值）。

4. 怎么用别人写好的函数？

- **最常见做法**：先导入别人写好的模块或库（比如用 `import xxx`），然后直接用它里面的函数（比如 `xxx.function()` 这样）。

5. 程序、逻辑、函数和函数调用之间的关系

- **程序**：就是一堆指令，按顺序让计算机干活。
- **逻辑**：程序里的思路或者套路，比如“如果怎样就怎样”这种判断。
- **函数**：帮你把复杂的、多次要用的操作打包成一个命令，就像一键启动。
- **函数调用**：就是你在需要的时候喊一声“微波炉，开！”——实质是用代码名（函数名）让函数干活。
- **总结一下**：一个程序离不开逻辑，而逻辑里常常要靠调用函数，整个流程就变得有条理、可复用。

课后练习

1. 如果你阅读的是本书的PDF版本，请使用Jupyter Notebook打开本文档。

2. 请把你最拿手的一道菜教给我！把每一步过程都按照输入、执行方法、结果的方式写出来。

3. 说明从学校到你家的路线，除了把每一步写清楚，还会在某些地方加上一些小提示（比如“路口这有个大广告牌，很显眼”“马路对面有条狗，别怕它”）。
   请用“输入、执行方法、结果”的方式，把这件事拆开，并指出哪些内容相当于“注释”。

4. 想象你要教别人用微波炉热饭。请将整个过程分解为“输入、执行方法、结果”，并说明“微波炉”在这里相当于什么。

5. 请用“网购”这件事，按“输入、执行方法、结果”来拆解一次完整的购物流程，并说明“函数调用”在这个场景中对应的是什么。

6. 假如你肚子饿了，让家人帮你做三明治。请把这个过程分成“输入、执行方法、结果”三步，并指出谁是“函数”，你是怎样“调用”这个函数的。

扩展知识

说起“交互艺术”，其实这东西一点不新鲜，早在20世纪60年代，一帮艺术家和科学家就开始琢磨：艺术作品能不能跟观众玩在一起，而不是你看我演、我画你看。后来随着电子技术、计算机普及，这事就更有意思了——比如装置艺术、感应雕塑、声音互动、到现在的数字投影和VR、AR，全都是交互艺术不断升级的产物。

除了Processing以外，还有不少技术也是做交互艺术、装置艺术的“老行家”——比如：

- **Arduino**（物联网和物理装置的好助手，能连各种传感器和马达）
- **OpenFrameworks**（C++语言的交互艺术开发利器，灵活、强大，适合视觉、声音、硬件互动）
- **TouchDesigner**（专门用来做实时互动和视觉效果的软件）
- **Unity/Unreal Engine**（这俩做游戏的东西现在常见于大型互动展和沉浸式体验）
- **Max/MSP/Jitter**（音视频信号处理和实时互动的“乐高积木”）

Processing官方资源和社区入口

建议你可以先逛逛Processing官网。这里有几个你一定用得上的好东西：

- **官方网站教程（Tutorials）**：在processing.org/learning/tutorials，有入门的、有进阶的，讲代码、讲视觉、讲交互，配图＋实例，跟着练很有收获。
- **教学案例（Examples）**：processing.org/reference里有，每一个语法点、每一个函数，几乎全带例子，打开就能运行。新手跟着抄一遍，基本上99%的小坑都能趟出来。
- **论坛和社区**：processing.org/community，以及Processing Discourse，国内还有processing吧和微信小群，遇到问题千万别憋着，上来就能找到“前辈队友”。

学习建议

其实，看流程代码、看艺术项目、查论坛问答，都是在给你“织知识的网”。
你可以把每个例子当一个“节点”：有的涉及视觉，有的涉及硬件，有的跟声音互动，把这些点串起来，你就能慢慢明白——交互艺术是怎么把代码、硬件、美、体验这么多东西揉在一起的。
而且，看社区答疑、跟别人交流，你能学到“解决问题的方法”——这比死记一个代码语法还值钱。
后面你每学一点新东西，都能马上找相关的扩展，比如从2D画圆改成3D画球，再加个声音或者鼠标互动，这就是你自己的“知识网络”，帮你查漏补缺、快速进阶。

说到编程语言发展，**变化很快，不必迷信某一种**。举个我的亲身经历：

- 高中时候，学的是**C语言**，那时啥都用它写，课本例子全靠它撑着。
- 到了大学，互联网流行了，**Java**一度成了主流，后台服务都用它。
- 我偏爱硬件、底层，选修了**C++**和**FPGA**，搞搞单片机、做做PCB。
- 工作后，游戏行业火爆，Unity成了宠儿，Unity用的就是**C#**。
- 后来智能手机铺天盖地，iOS用**Swift**，安卓玩**Dart**（Flutter），一转眼又是新世界。

要学明白、用明白，**不在于掌握无数“方言”**，关键是理解核心的编程思想，比如面向对象、流程控制、数据结构等等。
打个比方：你会用“做饭”的套路，无论是用锅、烤箱还是空气炸锅，你都能变着法儿做好吃的！

你可以随时访问 菜鸟教程、W3CSchool，用“面向对象程序设计”做关键词，挑本编程语言做练习，把基本套路学会，到哪种新语言都不会怕。
只要你掌握了“用代码建逻辑、表达需求、拆问题”的本事，**不管后面出来啥新语言、新技术，你都能举一反三。**

重点不是记住语法，是懂得背后思路和套路。 这样你未来遇到再新、再酷的艺术编程语言或交互方式，也都不怵，只会越学越快，越用越顺手。

练习题提示

以第五题为例，其他雷同：

- 输入：三明治的要求（比如要不要加蛋）
- 执行方法：你把需求告诉家人，家人操作，完成三明治
- 结果：你拿到三明治
- 家人就是三明治“函数”，你告诉他们怎么做（参数），他们帮你完成（函数调用）

第二章 绘制图形

本章内容

1. 计算机图形的基本组成元素是像素点
2. 计算机中存在屏幕坐标系、鼠标坐标系、世界坐标系等多种坐标系
3. RGBA（红、绿、蓝、透明度）构成了计算机屏幕的颜色
4. 学习使用Processing绘制基本的图形：点、线、矩形、椭圆等
5. 计算机中使用贝塞尔曲线来描述一条曲线
6. 学习使用Processing为图形设定填充色与边框色
7. 学习使用Processing绘制文本

1. 像素

如果你现在正在用 VS Code 打开并阅读 Jupyter Notebook 版本的这本教材，那么上一章的准备工作就说明已经成功完成。
接下来，你将正式踏上 Processing 的探索之旅。我们从一个极其基础的例子开始：请你拿出身边的一支笔，在纸上轻轻点一个点。看起来非常简单，对吧？
那么现在，我们加点难度。如果现在只允许你用一个个“点”，你该怎么在纸上写下数字“1”呢？或者说，更抽象一点，仅用点组合出一条直线？
我猜你的直觉一定是：不断地点，一列排列，让这些点依次排成一条直直的线。其实，这已经是绘图的基础逻辑了。
不过，如果我们想让这些点排列得够规整、够精确，你会发现靠肉眼对齐其实很困难。
这时，一个自然的想法就是——不如先在纸上画出一个大的网格，把纸切割成密密麻麻的小格子，然后你只需要把需要的位置的格子涂黑。
每个你点下去、涂黑的格子，其实就是一份小小的“像素”。
慢慢地，数字“1”也好、线条也好，甚至更复杂的图形，也都能通过选择性地“点亮”这些格子显现出来。
此时此刻，你手里的纸，已经变成了一张最原始的“坐标纸”。

让我们把这个想法再往前推进一步：假设这些格子变得极其细密，小到每个格子几乎看不见，这时你获得的其实正是计算机屏幕的本质——“像素网格”。
每一个像素，就是一个微小的格子，你可以单独决定它点亮还是熄灭、显示怎样的颜色。
只要你以此为基础，点亮该亮的，留白该留白，不知不觉中就能绘制出任何你想象中的图案，哪怕再复杂也没问题。

更有意思的是，一旦我们用坐标（比如 (x, y)）来精确标记每一个像素的位置，配合上 Processing 这样专注于可视化编程的工具，那么你其实已经实现了 Processing 里的核心命令之一：`point(x, y)`。
换句话说，从生活里最直接的“点一下纸”，到用一串代码直接“点亮屏幕上的某个坐标”，背后的逻辑是一以贯之的。

接下来我们面对的一个问题是，为什么你能看到坐标纸上的小格子，但却看不到屏幕上的像素？这就是PPI的概念。
PPI，全称 **Pixels Per Inch**，中文通常翻译为**每英寸像素数**，指的是设备显示屏或图像上每英寸（2.54厘米）所拥有的像素点数量。

可以想象“1 英寸”的直线上塞满了多少个小格子，每个格子就是一个像素，格子越多，图像就越精细。
你在手机或者电脑屏幕上，PPI 越高，每英寸范围内挤得下的点就越多，画面就越细腻，单个像素点就越不容易被肉眼察觉。“视网膜屏”其实就是在说 PPI 很高，高到人眼正常距离下看不清单个像素。
一张 800×600 像素的图片，如果在 100 PPI 的屏幕上显示，就差不多是 8 英寸 × 6 英寸的实际尺寸；
如果在 200 PPI 的屏幕上显示，就只会有 4 英寸 × 3 英寸那么大——因为像素密度翻倍，“像素”更密集，图片变小但更精细。

对于打印来说，PPI 决定了你的图片实际在纸上的大小和精度。
比如打印机通常要求 300 PPI 才能有杂志级别的精细输出，如果用 72 PPI 的图片去打印，出来的印刷品就会很模糊、颗粒感强。

举个实际例子 你手里有一台 1920×1080 像素的显示器，如果这个显示器的对角线是 24 英寸（16:9比例常见的尺寸），那它的 PPI 大概是多少？
$$ PPI = \frac {屏幕对角线像素数}{屏幕对角线英寸} \sqrt{1920^2 + 1080^2} \approx 2202 PPI = 2202 / 24 ≈ 92 $$
所以这台显示器的 PPI 是约 92——属于“正常”清晰度。
手机、平板常常高达 300 PPI 甚至 400 PPI 以上。

- 名词汇总
  - **像素**：
    像素（Pixel）是构成数字图像或屏幕显示的最小单位。**像素**没有大小
  - **PPI**：
    描述的是屏幕或图片的**像素密度**，用于数字图像和显示器。
  - **DPI**（Dots Per Inch）：
    更常用于打印机，指的是打印机每英寸能打印多少个点，受喷墨物理极限影响。
  - **分辨率**：
    常说的“1920×1080”，指的是一共横向和纵向有多少行多少列像素，不涉及密度。

现在，你已经掌握了数字绘图最本质的思路。
接下来，Processing 这套工具就可以帮你把想法变成代码，再让代码变成真正流动的视觉作品。

In [22]:

```
import py5

def setup(): # setup()函数在程序开始时执行一次，通常用于初始化设置
    py5.size(400, 400)  # 设置画布大小为400x400像素

    py5.stroke_weight(10)
    py5.point(50, 50)  # 绘制一个点，位置在(200, 200)，大小为10像素

    # 连续绘制10个点，形成一条线
    for i in range(20):  # 循环20次
        py5.point(50 + i * 5, 100)

    # 绘制多个点，形成一个园
    for i in range(100):
        angle = py5.TWO_PI * i / 100  # 计算每个点的角度
        x = 200 + 50 * py5.cos(angle)  # 计算x坐标
        y = 200 + 50 * py5.sin(angle)  # 计算y坐标
        py5.point(x, y)  # 绘制点

def draw(): # draw()函数在setup()之后循环执行，通常用于绘制动画或动态效
    pass

py5.run_sketch()  # 启动Py5程序，执行setup()和draw()函数
```

2. 计算机中的坐标系

在高中数学里，我们经常用“坐标点”精确描述一个位置，比如点 (50, 60) 代表：从原点 (0, 0) 出发，向右走 50、再往上走 60。类似地，点 (-50, -60) 意思是向左走 50，再往下走 60。

回忆象限和方向：
在我们熟悉的数学“笛卡尔坐标系”中：

(50, 60)：x、y 都为正，位于第一象限，在原点的右上方。
(-50, -60)：x、y 都为负，位于第三象限，在原点的左下方。

第二个问题：谁更靠上？
如果只考虑数学坐标系，坐标 (50, 60) 和 (100, 100)，很明显 Y 值越大越“上”，所以 (100,100) 比 (50,60) 更靠上。

但是！
当你到计算机世界，情况就变了。屏幕坐标和数学坐标有着非常大的不同：

在计算机屏幕里，左上角是原点 (0,0)。
向右是 X 轴正方向，向下是 Y 轴正方向！
X 越大越右，Y 越大反而离屏幕下方越近。
那么，当你写代码去绘制点 (50, 60) 与点 (-50, -60) 时会发生什么？

你会发现：

点 (50, 60) 会出现在屏幕内，距离左上角 50 个像素向右、60 个像素向下。
点 (-50, -60) 将完全不可见！因为坐标负值意味着超出了左上角以外的区域——在屏幕之外。
对于 (50, 60) 和 (100, 100)，由于 Y 较小的点在屏幕“更靠上”，所以 (50,60) 的位置“比” (100,100) 更靠上方，而不是下方！
很多同学第一次画图时会直觉地犯错，以为“Y值越大越高”——很遗憾，计算机的屏幕可不是这么理解的。

In [23]:

```
import py5

def setup(): # setup()函数在程序开始时执行一次，通常用于初始化设置
    py5.size(400, 400)  # 设置画布大小为400x400像素

    py5.stroke_weight(10)
    py5.stroke(255, 0, 0)  # 设置描边颜色为红色
    py5.point(50, 60)  # 绘制一个点，位置在(50, 60)，大小为10像素

    py5.stroke(0, 255, 0)  # 设置描边颜色为绿色
    py5.point(-50, -60)  # 绘制一个点，位置在(-50, -60)，大小为10像素

    py5.stroke(0, 0, 255)  # 设置描边颜色为蓝色
    py5.point(100, 100)  # 绘制一个点，位置在(100, 100)，大小为10像素


def draw(): # draw()函数在setup()之后循环执行，通常用于绘制动画或动态效
    pass

py5.run_sketch()  # 启动Py5程序，执行setup()和draw()函数
```

在我们实际进行计算机图形绘制时，会面对许多中坐标系统：屏幕坐标系，鼠标坐标系，如果你未来进行游戏制作，还会面对相对坐标系和世界坐标系。

- 屏幕坐标：
  就像你看到的窗口，左上角（0,0）是屏幕原点，每一个像素都能用(x, y)描述，比如 (80, 55) 就在左上方往右80、下55的像素点那里。

- 鼠标坐标：
  任何时候你移动鼠标，计算机都能告诉你“此刻鼠标指的是什么屏幕坐标点”，比如 `py5.mouse_x = 400, mouseY = 200` 表示鼠标正处于距离屏幕左上各400、200的位置。

- 相对坐标（局部坐标）：
  假如你写个游戏角色，把角色身体当成新原点，则角色头发、左手、鞋子各自的位置，都是相对于“身体中心”设定的。比如头顶是 (0, 80)，鞋子 (0,-80)。

- 世界坐标：
  在大型游戏或虚拟宇宙时，我们常常不仅有相对坐标，还要虚拟出“世界空间”，比如以地球中心为 (0,0,0)，你的位置可能是 (127, -55, 48)。 一切复杂对象、人物之间的运动、物理运算，都是建立在“世界坐标”上，只有最终渲染才映射成屏幕坐标。
  而此时角色头发、左手、鞋子就拥有至少两套坐标系统，比如头顶（0,80）和（127， -55 + 80， 48）

- 名词汇总

  - 屏幕坐标系：左上角原点，Y 越大越下。
  - 数学坐标系：中心原点，Y 越大越上。
  - 相对坐标：局部基点为原点（比如角色身体、物体局部）。
  - 世界坐标：全局场景绝对位置。
  - 鼠标坐标：每次移动都能获知相对于屏幕左上角的像素点。

In [30]:

```
import py5
import math

def setup():
    py5.size(800, 600)
    my_font = py5.create_font("Noto", 12)
    py5.text_font(my_font)
    py5.text_align(py5.LEFT, py5.TOP)
    py5.rect_mode(py5.CORNER)
    py5.no_stroke()

def draw():
    py5.background(250)
    # ====== 屏幕坐标网格 ======
    py5.stroke(225)
    for x in range(0, py5.width, 50):
        py5.line(x, 0, x, py5.height)
```

```
for y in range(0, py5.height, 50):
    py5.line(0, y, py5.width, y)
py5.no_stroke()

# ===== 屏幕坐标举例点 =====
py5.fill(60, 130, 210)
py5.rect(100, 100, 60, 40)
py5.fill(255)
py5.text("屏幕坐标: (100,100)", 102, 102)

# ===== 鼠标屏幕坐标 =====
py5.fill(0)
py5.text(f"鼠标屏幕 (mouse): ({py5.mouse_x}, {py5.mouse_y})", 10, 10)
py5.fill(220, 100, 100)
py5.ellipse(py5.mouse_x, py5.mouse_y, 12, 12)

# ====== 相对坐标/归一化坐标 ======
norm_x = py5.mouse_x / float(py5.width)
norm_y = py5.mouse_y / float(py5.height)
py5.fill(0)
py5.text(f"归一化(0-1): (%.2f, %.2f)" % (norm_x, norm_y), 10, 36)
py5.fill(60, 220, 100)
py5.ellipse(norm_x * py5.width, norm_y * py5.height, 10, 10)
py5.text("中心为(0.5,0.5)", py5.width * 0.5 + 8, py5.height * 0.5 + 8)
py5.stroke(60, 220, 100, 128)
py5.line(py5.width * 0.5, 0, py5.width * 0.5, py5.height)
py5.line(0, py5.height * 0.5, py5.width, py5.height * 0.5)
py5.no_stroke()

# ===== 世界坐标变换区 =====
world_pos = (550, 300)
world_angle_deg = 30
world_angle_rad = math.radians(world_angle_deg)
world_grid_span = 200
world_grid_step = 40

with py5.push_matrix():
    # 设置世界坐标原点与旋转
    py5.translate(world_pos[0], world_pos[1])
    py5.rotate(world_angle_rad)

    # ==== 绘制“世界坐标”网格 ====
    py5.stroke(180, 200, 255)
    for x in range(-world_grid_span, world_grid_span + 1, world_grid_step):
        py5.line(x, -world_grid_span, x, world_grid_span)
    for y in range(-world_grid_span, world_grid_span + 1, world_grid_step):
        py5.line(-world_grid_span, y, world_grid_span, y)
    # 世界坐标原点与主轴
    py5.stroke(60, 130, 210)
    py5.stroke_weight(2)
    py5.line(0, 0, 60, 0)
    py5.line(0, 0, 0, 60)
    py5.no_stroke()
    py5.fill(60, 130, 210)
    py5.ellipse(0, 0, 14, 14)
    py5.fill(30)
    py5.text("世界原点 (0,0)", 10, -24)
    py5.text("X", 62, -12)
    py5.text("Y", -14, 62)

    # ==== 在世界坐标固定位置画个对象 ====
    py5.fill(180, 90, 255, 180)
    py5.rect_mode(py5.CENTER)
    py5.rect(80, 50, 40, 40)
    py5.fill(30)
```

```
        py5.text("世界坐标 (80,50)", 80 + 12, 50 - 8)
        py5.rect_mode(py5.CORNER)

    # ==== 鼠标在世界坐标区的映射 ====
    # 逆变换：mouse坐标 → 世界坐标（逆旋转+逆平移）
    dx = py5.mouse_x - world_pos[0]
    dy = py5.mouse_y - world_pos[1]
    c = math.cos(-world_angle_rad)
    s = math.sin(-world_angle_rad)
    world_mouse_x = dx * c - dy * s
    world_mouse_y = dx * s + dy * c
    with py5.push_matrix():
        py5.translate(world_pos[0], world_pos[1])
        py5.rotate(world_angle_rad)
        # 显示鼠标在世界的点（红色）
        py5.fill(220, 60, 80)
        py5.ellipse(world_mouse_x, world_mouse_y, 16, 16)
        py5.fill(0)
        py5.text(
            f"鼠标的世界坐标: (%.1f, %.1f)" % (world_mouse_x, world_mouse_y),
            world_mouse_x + 18, world_mouse_y - 8
        )
    # ====== 信息区 ======
    py5.fill(80)
    py5.text(
        "□□ 屏幕、鼠标、点阵\n"
        "➡□ 世界坐标系及变换\n"
        "下方为归一化/相对坐标\n"
        "\n拖动鼠标观察变化",
        22, py5.height - 110
    )

py5.run_sketch()
```

3. 颜色系统

你有没有想过，我们为什么能看到彩色的世界？其实这都得感谢我们的眼睛——可以说，“人眼就是一台会自适应的摄像头”。
不过它的魔力并不是什么都能看，主要只对特定的“可见光”有感觉。这段光波大概在400到700纳米之间，也就是红、绿、蓝为主的“感觉区”。
你眼球里有三种专门感光的小细胞，各自偏爱某种颜色（红、绿、蓝），咱们见到的丰富颜色，其实都是靠它们按比例组合的。
就像调色盘，红绿蓝一混合，能变出海量的颜色。
这么一说，你可能觉得自己的眼睛和数码相机挺像，都有感光元件在专心收集光线信号，然后用“混色”组合成漂亮的画面。

那这些光是怎么来的呢？其实有两种“光源”在给我们世界点亮。

- 第一种是自发光，比如太阳、灯泡、LED——它们自己就“放电”造光。这类光最直接，永远是能量满满地往外发射。
- 第二种就是反光，像你手里的书、墙上的画、地上的瓷砖，很多东西它们本身不会发光，但它们会把别的地方来的光反弹给你。

有趣的是，我们看到的颜色，其实只是这些物体“挑着性子”反射了一些色光，把另外的一部分都悄悄吸收掉。
比如红苹果，它可不是自己发红光，而是白光照射后只反弹出红色那部分进了你眼睛。
说到底，如果没有光，这一切都归于黑暗，所有形状、颜色都不存在，好比电视关机，啥都没啦。

说到咱们生活用品的颜色系统，这就跟实际的“发光”和“反光”密不可分。
比如家里的书、纸张、杂志这种，叫反光系统。
它们靠的是CMYK方法来“混色”——其实就是青色、品红、黄色和黑色，这几种颜色的墨水相互配合，能印出大部分咱们常见的纸上色彩。
但你别看印刷品五彩缤纷，其实它的色域有限，有些鲜艳的蓝色、荧光绿纸张上怎么都调不出来，见过杂志和电脑屏幕同一幅图的色差没？这就是CMYK的局限性。

再聊屏幕。手机、电视、显示器这些都是自发光系统。
每个像素本质上就是红、绿、蓝三种细小灯泡，只要分别调亮度，就能混出超级丰富的颜色。
有了这招，电脑屏幕能显示很多种接近真实世界的鲜艳颜色。
像咱们编程常用的“RGB色域”，对于绝大多数用途来说已经够用了。
不过普通显示器的色域也有限（基本上都是sRGB标准），如果追求电影级、设计级效果，还得上广色域显示器，比如AdobeRGB或者DCI-P3。
有一说一，直接发光的系统色彩范围（尤其是一些专业激光影院）是天花板，远远超过一般的纸上和普通屏幕。
可惜日常生活里，色域最大的只有想象力和太阳啦！

在计算机的数字世界丽，RGBA颜色模式是必不可少的“调色盘”。
咱们最常见的是RGB——红、绿、蓝三种通道，每个通道通常取值从0到255，调来调去就能拼出各种颜色。
比如写 (255, 0, 0) 就是最纯正的红，(0, 255, 0) 是不掺杂任何其他色的绿，看起来特别直观。
不过现实世界可不光只有颜色，有层次、有穿透、有叠加，才真实又美观，这时候A通道就登场啦！
这A，是Alpha，管的是“透明度”，让画面上的每一个像素还能有0到255（或者0.0到1.0）这样的透明等级。
比如py5里 fill(200, 50, 150, 128)，就是让你画一个半透明的紫红色圆。
如果想表达一个带点透明感的绿色，可以写成rgba(128, 200, 40, 0.7)，那就是70%的不透明，仿佛水彩一样可以叠加出层次感。

至于，为何使用0~255这个范围表示颜色，简单来说默认的RGBA系统使用8位2进制数表示一种单一的颜色。 n位2进制数可以表示2的n次方个10进制数，关于数的进制问题，会在后面的章节中详细讲解。

- 名词汇总
  - **反光系统：**
    纸张、画册这类靠环境光照亮再“反弹”一些色光到你眼里，自己不发光的显示方式。
  - **自发光系统：**
    像电视或手机那样自己能发光的屏幕，通过红绿蓝三色点直接制造各种颜色。
  - **色域：**
    设备能表现出来的颜色范围，好比“色彩的地盘”，地盘越大能显示的颜色越丰富。
  - **RGBA：**
    数码色彩的四要素，红绿蓝控制颜色，A控制透明度，多用于编程和设计。

In [39]:

```
import py5

# 初始化滑块位置
def setup():
    py5.size(600, 400)
    py5.background(255, 0, 0) # 设置背景颜色为红色

    py5.stroke_weight(10) # 设置描边宽度为10像素

    py5.stroke(0, 255, 0) # 设置描边颜色为绿色
    py5.point(50, 50) # 绘制一个点，位置在(50, 50)

    py5.stroke(0, 0, 255) # 设置描边颜色为蓝色
    py5.point(100, 100) # 绘制一个点，位置在(100, 100)

def draw():
    pass

# 运行程序
py5.run_sketch()
```

In [ ]:

```
import py5

# 全局变量
r, g, b, a = 128, 128, 128, 255 # 初始灰色，不透明
active_slider = None # 当前被激活的滑块索引，0:R, 1:G, 2:B, 3:A

# 四个滑块的轨道位置和尺寸（x, y, 宽度, 高度）
slider_width = 300
slider_height = 20
sliders = [] # 每个元素是(x, y, width, height)，以及一个指向值的引用？实际上我们会用索引访问全局变量

# 初始化滑块位置
def setup():
    py5.size(600, 400)
    # 设置每个滑块的位置
    global sliders
    slider_x = (py5.width - slider_width) // 2 # 水平居中
    # 四个滑块垂直排列，顶部在预览矩形下方
    preview_bottom = 200 + 20 # 预览矩形底部再往下20像素
    slider_y0 = preview_bottom + 20 # 第一个滑块顶部位置
    gap = 40 # 每个滑块之间的垂直间距

    sliders = [
        (slider_x, slider_y0 + 0*gap, slider_width, slider_height), # R
        (slider_x, slider_y0 + 1*gap, slider_width, slider_height), # G
        (slider_x, slider_y0 + 2*gap, slider_width, slider_height), # B
        (slider_x, slider_y0 + 3*gap, slider_width, slider_height)  # A
    ]

# 绘制每一帧
def draw():
    py5.background(200) # 灰色背景

    # 绘制预览矩形
    py5.fill(r, g, b, a)
    py5.rect_mode(py5.CENTER)
    py5.rect(py5.width//2, 100, 400, 150) # 中心在(300,100)位置

    # 绘制四个滑块
    py5.rect_mode(py5.CORNER)
    for i, (x, y, w, h) in enumerate(sliders):
        # 绘制轨道（灰色轨道，上面有一条当前值的标记）
        py5.fill(100)
```

```
        py5.rect(x, y, w, h)

        # 根据当前值计算滑块头位置（将0-255映射到轨道的x到x+w）
        slider_value = [r, g, b, a][i]
        head_x = py5.remap(slider_value, 0, 255, x, x+w)
        # 绘制滑块头（小圆点或者小矩形）
        py5.fill(255)
        py5.rect(head_x-5, y-5, 10, h+10)  # 滑块头略高于轨道

        # 显示滑块标签和当前值
        labels = ['R', 'G', 'B', 'A']
        py5.fill(0)
        py5.text_size(16)
        py5.text(f"{labels[i]}: {slider_value}", x-60, y+h/2+5)  # 在滑块左侧显示

# 鼠标按下事件
def mouse_pressed():
    global active_slider
    # 检查是否按在某个滑块头上（或轨道上）
    for i, (x, y, w, h) in enumerate(sliders):
        # 获取当前滑块值对应的滑块头位置
        slider_value = [r, g, b, a][i]
        head_x = py5.remap(slider_value, 0, 255, x, x+w)
        # 创建一个滑块头的矩形区域（扩展一些，容易点中）
        head_rect = (head_x-10, y-10, 20, h+20)
        # 或者简单点，检查鼠标是否在轨道矩形内（包括轨道和滑块头）
        # 我们检查鼠标在轨道矩形内就算点击了该滑块
        if (x <= py5.mouse_x <= x+w) and (y <= py5.mouse_y <= y+h):
            active_slider = i
            # 同时更新值，因为点击轨道任意位置也要更新
            update_slider(i)
            return

# 更新当前活动滑块的值（根据鼠标位置）
def update_slider(index):
    global r, g, b, a
    x, y, w, h = sliders[index]
    # 将鼠标x坐标限制在轨道的x到x+w之间，并映射到0-255
    new_value = py5.remap(py5.mouse_x, x, x+w, 0, 255)
    new_value = py5.constrain(new_value, 0, 255)

    # 更新对应变量
    if index == 0:
        r = int(new_value)
    elif index == 1:
        g = int(new_value)
    elif index == 2:
        b = int(new_value)
    elif index == 3:
        a = int(new_value)

def mouse_dragged():
    if active_slider is not None:
        update_slider(active_slider)

def mouse_released():
    global active_slider
    active_slider = None

# 运行程序
py5.run_sketch()
```

4. 绘制简单图形

In [ ]:

```
import py5

def setup():
    py5.size(800, 400)
    my_font = py5.create_font("Noto", 16)
    py5.text_font(my_font)
    py5.text_align(py5.LEFT, py5.TOP)

    py5.background(240)

    # 把画布分成两排，每排4格，图形居中分布
    cell_w = py5.width // 4
    cell_h = py5.height // 2

    # 1. 画点
    # 使用stroke_wright()设置点的大小
    # 使用point()函数绘制点
    py5.stroke(0)
    py5.stroke_weight(8)  # 点粗一点方便看
    px, py_ = cell_w//2, cell_h//2
    py5.point(px, py_)
    py5.stroke_weight(1)  # 恢复线宽
    py5.fill(0)
    py5.text("点 point()", px-20, py_ + 40)

    # 2. 画直线
    # 使用stroke()设置线条颜色
    # 使用stroke_weight()设置线条宽度
    # line参数说明
    # x1, y1, x2, y2 分别是线条的起点和终点坐标
    # 这里的线条是从左上角到右下角
    # 也可以使用lines()函数绘制多条线
    py5.stroke(255, 0, 0)
    lx = cell_w + cell_w//2
    ly1 = cell_h//2 - 20
    ly2 = cell_h//2 + 20
    py5.line(lx, ly1, lx, ly2)
    py5.fill(255, 0, 0)
    py5.text("直线 line()", lx-30, ly2 + 30)

    # 3. 画三角形
    # 自己阅读py5的文档，了解triangle()函数的用法
    py5.stroke(0, 128, 0)
    tx = cell_w*2 + cell_w//2
    ty = cell_h//2
    py5.fill(0, 180, 0, 100)  # 半透明绿
    py5.triangle(tx, ty-30, tx-30, ty+30, tx+30, ty+30)
    py5.fill(0)
    py5.text("三角形 triangle()", tx-50, ty + 60)

    # 4. 画矩形
    # 自己阅读py5的文档，了解rect()函数的用法
    # 此外，py5还提供了rect_mode()函数来设置矩形的绘制模式
    # 自行阅读py5的文档，了解rect_mode()函数的用法
    py5.stroke(0,0,255)
    rx = cell_w*3 + cell_w//2
    ry = cell_h//2
    py5.fill(0, 0, 255, 80)  # 半透明蓝
    py5.rect_mode(py5.CENTER)
    py5.rect(rx, ry, 60, 40)
    py5.rect_mode(py5.CORNER)  # 恢复默认
    py5.fill(0)
    py5.text("矩形 rect()", rx-40, ry + 40)

    # 5. 画正方形
```

```
    # 正方形也是矩形，只要宽高相等即可
    # 使用rect()函数绘制正方形
    py5.stroke(150, 0, 150)
    sx = cell_w//2
    sy = cell_h + cell_h//2
    py5.fill(180, 0, 180, 90) # 半透明紫
    py5.rect_mode(py5.CENTER)
    py5.rect(sx, sy, 50, 50) # 正方形，只要宽高相等
    py5.rect_mode(py5.CORNER)
    py5.fill(0)
    py5.text("正方形 rect() ", sx-35, sy+40)

    # 6. 画椭圆
    py5.stroke(0, 170, 170)
    ex = cell_w + cell_w//2
    ey = cell_h + cell_h//2
    py5.fill(0, 200, 200, 100)
    py5.ellipse(ex, ey, 70, 40)
    py5.fill(0)
    py5.text("椭圆 ellipse()", ex-40, ey+40)

    # 7. 画圆形
    # py5也提供了circle()函数来绘制圆形
    # 但这里我们使用ellipse()函数绘制圆形
    # 因为ellipse()函数可以绘制椭圆和圆形
    # 只要宽高相等即可绘制圆形
    # 注意：py5.circle()函数是Processing 4.0版本新增的
    # 在py5中，circle()函数是别名，实际上还是调用的ellipse
    py5.stroke(255, 200, 0)
    cx = cell_w*2 + cell_w//2
    cy = cell_h + cell_h//2
    py5.fill(255, 225, 60, 90)
    py5.ellipse(cx, cy, 50, 50) # 宽高一样就是圆
    py5.fill(0)
    py5.text("圆 ellipse()", cx-30, cy+40)

    # 8. 画弧线 (半圆)
    py5.stroke(255,128,0)
    ax = cell_w*3 + cell_w//2
    ay = cell_h + cell_h//2
    py5.no_fill()
    py5.arc(ax, ay, 60, 60, 0, py5.PI) # 0到PI画半圆弧，默认OPEN模式
    py5.fill(0)
    py5.text("弧线 arc()", ax-32, ay+45)

def draw():
    pass

py5.run_sketch()
```

5. 贝塞尔曲线

有了像素点的概念后，我们理论上就可以绘制各种图片。
但实际上，想象一下这样一个场景：比如你在设计一个可爱的笑脸图标。
如果我们用像素点，一点一点去“拼”出这个笑脸，那么在不同分辨率的屏幕上，比如小手机屏还是超大显示器，我们就得分别准备不同大小的像素图。
而且，随着分辨率的提升，像素点的数量会狂飙，图片文件也会越来越大，非常占用空间。
更可怕的是，要是把一张小图片硬拉大，笑脸边缘就会变成锯齿状，像马赛克一样，一点都不圆润，这就是我们常说的“像素化”或者“失真”。

- 计算机中有一种算法称为“抗锯齿”，就是在解决像素曲线这个问题

为了解决存储空间占用越来越大这个问题，聪明的工程师们想到了一个特别妙的办法——用贝塞尔曲线来描述图形。

贝塞尔曲线最初是由法国工程师皮埃尔·贝塞尔（Pierre Bézier）在上世纪60年代为雷诺汽车公司设计汽车外形的时候发明的。
后来被计算机领域的专家们引入到了计算机绘图的过程中，成为了矢量图形的基础。

后来被计算机领域的专家们引入到了计算机绘图的过程中，成为了矢量图形的基础。
现在，像我们日常看到的各种字体、App里的小图标、动画里的曲线路径，背后都有贝塞尔曲线的身影！

- 矢量图：利用**数学描述**表示图形
- 位图：利用**像素点**描述表示图形

贝塞尔曲线通过几个关键的点——我们叫它们“控制点”——来确定一条曲线的形状。
比如说，最简单的一次贝塞尔曲线就像直线一样，只需要起点和终点。
加上一个控制点，就是二次贝塞尔曲线，能画出简单的弯曲；
如果再多一个控制点，就是三次贝塞尔曲线，可以画出又酷又优雅的复杂曲线。
你可以把这些控制点想象成几根“看不见的橡皮筋”，它们拉着曲线不断变弯、变直，让图形变得超级丝滑！

生成过程——通过“拉扯控制点”一步步长出曲线

1. 先定几个点（叫控制点），比如 $P_0, P_1, P_2,\dots$。
2. 画虚线把这些点连起来，但别急，这还不是最终曲线。
3. 想象有一只小虫，从起点慢慢爬到终点，在每一瞬间，这只“虫”并不是一直走直线，而是不断参照所有控制点，躲着、靠近、被“拉”着向不同方向。
4. 这个动态过程，可以用数学公式或者递归算法来描述，得到每个时间点（t）的位置坐标。

通用表达式（不要怕，看懂意思就行！）

对于 n 阶贝塞尔曲线，有 $n+1$ 个控制点 $P_0, P_1, \dots, P_n$，
t 取值在 0 到 1，表示沿曲线的进度。

曲线上的点坐标 $B(t)$ 要这样算：

$$ B(t) = (1-t)^n P_0 + n(1-t)^{n-1} t P_1 + \dots + t^n P_n $$

常见阶数举例

- **1阶贝塞尔线（直线）：**
  $B(t) = (1-t) P_0 + t P_1$（简单插值）
- **2阶贝塞尔曲线（二次）：**
  $B(t) = (1-t)^2 P_0 + 2(1-t)t P_1 + t^2 P_2$
- **3阶贝塞尔曲线（三次）：**
  $B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t) t^2 P_2 + t^3 P_3$

其实概念很像“加权平均”——t越靠近0，越偏重前面的点；t越靠近1，越偏后面。

德卡斯特里奥算法——贝塞尔曲线的“动画拉面法”

这是一个非常直观、几何化的计算贝塞尔曲线坐标的方法，具体可以像搭积木一样理解：

1. **第一步**：把控制点全部两两相连，得到新的中间点。
2. **第二步**：再把新的中间点，继续两两相连，得到更少、更靠内部的点。
3. **一直递归下去**，直到最后剩一个点。
4. **这个唯一的点，就是曲线在某个时刻t下的实际位置。**

为什么可以用递归进行运算？我们把之前的曲线点公式稍微写长一些

$$ B(t) = (1-t)^n t^0 P_0 + (1-t)^{n-1} t^1 P_1 + (1-t)^{n-2} t^2 P_2 + (1-t)^{n-3} t^3 P_2 + \dots + (1-t)^3 t^{n-3} P_{n-3} + (1-t)^2 t^{n-2} P_{n-2} + (1-t)^1 t^{n-1} P_{n-1} + (1-t)^0 t^n P_n $$

现在开始两两组合

$$ B(t) = (1-t)^{n-1}[(1-t) P_0 + t P_1] + (1-t)^{n-3}t^2[(1-t) P_2 + t P_2] + \dots + (1-t)^2 t^{n-3} [(1-t) P_{n-3} + t P_{n-2}] + t^{n-1}[(1-t) P_{n-1} + t P_n] $$

你可以继续试着两两相加的方式推到下去，当你推导至最后一层时，就只需要计算两个点即可，这就是递归的概念。

- 名词汇总
  - **贝塞尔曲线：**
    用一组控制点决定走向的平滑曲线，像是被点“拉”出来的橡皮筋线条。

In [8]:
```
import py5
```

```
# -- 定义每阶贝塞尔曲线的控制点与区域属性 --
regions = [
    {  # 1阶：两点一线（直线）
        'points': [(60, 320), (320, 60)],
        'color': '#FF3333',
        'order': 1,
        't': 0,
    },
    {  # 2阶：三点二次曲线
        'points': [(60, 320), (190, 80), (320, 320)],
        'color': '#33CC66',
        'order': 2,
        't': 0,
    },
    {  # 3阶：四点三次曲线
        'points': [(60, 320), (120, 60), (260, 80), (320, 320)],
        'color': '#3366FF',
        'order': 3,
        't': 0,
    },
    {  # 4阶：五点四次曲线
        'points': [(60, 320), (100, 60), (160, 340), (260, 80), (160, 60)],
        'color': '#FF33CC',
        'order': 4,
        't': 0,
    },
]

region_size = 380  # 每块区域的大小
margin = 10        # 区域与画布或相邻块之间的边距

# -- 德卡斯特里奥算法，递归计算贝塞尔曲线某t点的坐标 --
def de_casteljau(t, pts):
    # pts: 控制点列表，每个元素是(x, y)
    if len(pts) == 1:
        return pts[0]
    new_pts = []
    for i in range(len(pts)-1):
        x = (1-t)*pts[i][0] + t*pts[i+1][0]
        y = (1-t)*pts[i][1] + t*pts[i+1][1]
        new_pts.append((x, y))
    return de_casteljau(t, new_pts)

def setup():
    py5.size(region_size*2 + margin*3, region_size*2 + margin*3)
    py5.background(250)
    my_font = py5.create_font("Noto", 16)
    py5.text_font(my_font)
    py5.text_align(py5.LEFT, py5.TOP)

def draw():
    py5.background(250)
    steps = 120  # 曲线采样点数，越大越平滑

    for i, region in enumerate(regions):
        # -- 计算该区域左上角坐标 --
        rx = margin + (i % 2) * (region_size + margin)
        ry = margin + (i // 2) * (region_size + margin)
        # -- 到区域内部绘图 --
        py5.push_matrix()
        py5.translate(rx, ry)

        # 区域背景&边框
        py5.no_stroke()
        py5.fill(245)
```

```
        py5.rect(0, 0, region_size, region_size, 20)
        py5.stroke(200)
        py5.no_fill()
        py5.rect(0, 0, region_size, region_size, 20)

        # -- 画虚线控制多边形 --
        pts = region['points']
        py5.stroke(120, 120, 120, 150)
        py5.stroke_weight(1)
        for j in range(len(pts)-1):
            px1, py1 = pts[j]
            px2, py2 = pts[j+1]
            py5.line(px1, py1, px2, py2)

        # -- 画控制点 --
        py5.fill(0)
        py5.stroke(40)
        for j, (px, py_) in enumerate(pts):
            py5.ellipse(px, py_, 12, 12)
            # 控制点编号
            py5.fill(70)
            py5.text_size(14)
            py5.text(f'P{j}', px + 8, py_ - 20)
            py5.fill(0)

        # -- 动态画贝塞尔曲线的“已经写出”部分 --
        py5.stroke(region['color'])
        py5.stroke_weight(3)
        py5.no_fill()
        py5.begin_shape()
        t_now = region['t']
        for s in range(steps+1):
            t_val = s * t_now / steps
            x, y = de_casteljau(t_val, pts)
            py5.vertex(x, y)
        py5.end_shape()

        # -- 动态t点的拖尾小球 --
        if t_now > 0:
            bx, by = de_casteljau(t_now, pts)
            py5.no_stroke()
            py5.fill(region['color'])
            py5.ellipse(bx, by, 16, 16)

        # -- 区域标题（阶数）/解释说明 --
        py5.fill(60)
        py5.text(f'{region["order"]}阶贝塞尔', 20, 15)
        py5.text(f'控制点数： {region["order"]+1}', 20, region_size - 34)

        py5.pop_matrix()

        # -- t动画: 匀速递增，画完再重置 --
        region['t'] += 0.009
        if region['t'] > 1:
            region['t'] = 0

def key_pressed():
    # 按下任意键重置动画
    for region in regions:
        region['t'] = 0

py5.run_sketch()
```

6. 为图形添加边线与填充色

In [13]:

```
import py5

def setup():
    py5.size(800, 600)
    my_font = py5.create_font("Noto", 16)
    py5.text_font(my_font)
    py5.text_align(py5.LEFT, py5.TOP)
    py5.rect_mode(py5.CENTER) # 以中心点作为矩形基准点

def draw():
    py5.background(240)
    start_x, start_y = 140, 80      # 网格左上角第一个中心点位置
    grid_w, grid_h = 220, 220       # 每个展示区
    w, h = 100, 60                  # 矩形本身大小
    py5.no_stroke()

    # --------- 1. 默认黑描边 灰色填充 ----------
    py5.push_style() # 保存当前样式
    py5.stroke(0) # 阅读py5文档了解stroke()函数
    py5.fill(180) # 阅读py5文档了解fill()函数
    py5.rect(start_x, start_y, w, h)
    py5.fill(0)
    py5.text("1. 默认 黑描边+灰填充\nstroke(0); fill(180)", start_x-w/2, start_y+h/2+16)
    py5.pop_style() # 恢复样式

    # --------- 2. 只有红色描边（无填充） ----------
    py5.push_style()
    py5.stroke(255,0,0)
    py5.no_fill() # 阅读py5文档了解no_fill()函数
    py5.rect(start_x+grid_w, start_y, w, h)
    py5.fill(150,0,0)
    py5.text("2. 只有红描边\nstroke(255,0,0); no_fill()",
            start_x+grid_w-w/2, start_y+h/2+16)
    py5.pop_style()

    # --------- 3. 只有绿色填充（无描边） ----------
    py5.push_style()
    py5.no_stroke() # 阅读py5文档了解no_stroke()函数
    py5.fill(0, 200, 90)
    py5.rect(start_x+2*grid_w, start_y, w, h)
    py5.fill(0,100,30)
    py5.text("3. 只有绿填充\nno_stroke(); fill(0,200,90)",
            start_x+2*grid_w-w/2, start_y+h/2+16)
    py5.pop_style()

    # --------- 4. 半透明蓝色填充，黑描边 ----------
    py5.push_style()
    py5.stroke(0)
    py5.fill(80, 120, 255, 120) # 半透明
    py5.rect(start_x, start_y+grid_h, w, h)
    py5.fill(30,60,160)
    py5.text("4. 半透明蓝色\nfill(80,120,255,120)",
            start_x-w/2, start_y+grid_h+h/2+16)
    py5.pop_style()

    # --------- 5. 带透明度描边 ----------
    py5.push_style()
    py5.stroke(200,0,0, 100)   # 红色半透明
    py5.no_fill()
    py5.stroke_weight(8)
    py5.rect(start_x+grid_w, start_y+grid_h, w, h)
    py5.no_stroke()
    py5.fill(150,0,0)
```

```
    py5.text("5. 半透明红描边\nstroke(200,0,0,100)",
            start_x+grid_w-w/2, start_y+grid_h+h/2+16)
    py5.pop_style()

    # --------- 6. 设置描边粗细 -------------------------
    py5.push_style()
    py5.stroke(80,60,200)
    py5.stroke_weight(12)
    py5.no_fill()
    py5.rect(start_x+2*grid_w, start_y+grid_h, w, h)
    py5.no_stroke()
    py5.fill(93,80,120)
    py5.text("6. 粗描边\nstroke_weight(12)",
            start_x+2*grid_w-w/2, start_y+grid_h+h/2+16)
    py5.pop_style()

py5.run_sketch()
```

7. 文本绘制

文字最开始是为了让人们**记录信息、表达思想**。远古时代，人们通过画符号、刻图案，把语言或事件记录下来。
后来，人类社会变复杂，口头交流不够用了，就发明了结构化的“文字”。于是不同文明，有了自己的书写方式。
为了让更多人读懂、方便传播，人们开始在“点、线、结构”上改良文字样子，于是就有了**各种“字体”或“字形”**。
字体其实就是“同一套文字，不同的外观风格”，好比穿衣服的不同风格。

在**古代中国**，人们使用毛笔蘸墨写字。毛笔的软毫粗细、运笔快慢，都让汉字变得千姿百态。
为了在不同场合、不同媒介上写出漂亮或实用的字，中国历史上产生了许多著名字体，比如**篆书**（古老典雅）、**隶书**（平整易认）、**楷书**（工整规范）、**行书**（流畅自然）、**草书**（随性狂放）。
这些字体，既有实用性，也有极高的艺术性。要写好一个字，需要反复练习、体会“点横撇捺”的奥妙。

在**旧欧洲时代**，主流书写工具是羽毛笔（鹅毛笔）。羽毛笔头又粗又窄，墨水容易流淌，不同的书写角度创造了许多独有的拉丁字母风格。
比如，中世纪的“哥特体”（Gothic Script），字母轮廓鲜明、厚重有力，是因为羽毛笔的横与竖不同宽度；
后来的圆体（Humanist）、斜体（Italic）等，也都是为了美观、快速书写或适应书印技术发展而产生的。

数字时代，文字从纸面搬到了屏幕上。
计算机中的文字本质上是许多曲线的组合。比如，"A"字，人眼看是线，其实计算机用一堆简单曲线（如直线、贝塞尔曲线）拼成那个样子。
人们仍然希望在屏幕表达不同风格的文字，所以在**数字世界里，同样需要各种字体**——既可以表达美观，也能适合不同用途（比如报纸、网页、书法字、LOGO等）。

一开始，计算机屏幕只显示26个英文字母和常用标点（比如最早的ASCII字符），每个字母样子都直接画成一堆像素点，全部塞进内存，这被叫做**“点阵字体”**（bitmap font）。
这种方式简单，比如数字“8”就是7×5格子的点阵。但它有两个问题——只能预设少量符号，而且一放大就马赛克。

后来文字数量增加，有了多国语言（比如中文成千上万个汉字、日文、韩文等），仅靠像素点阵，就会占用巨量存储，还没法随意变大变小，于是有了**“矢量字体”**。（即用数学曲线——比如贝塞尔曲线描述每个字形）

这样，每种字体都需要一个**专门的“字体文件”**，里面包含了每个字母/汉字的详细描画命令。程序要显示文字时，就到字体文件查一查，把曲线渲染到屏幕上。
如果你使用了错误的字体，而实际文字又不在该字体支持的字符集中，程序就无法找到字形，只能显示乱码方块（有的叫“豆腐字”），或者用问号代替。

常见字体文件格式

- **TrueType Font（.ttf）**：最常见，可跨平台。
- **OpenType Font（.otf）**：功能更强，支持高级排版。
- **Web字体格式**（.woff, .woff2）：适合网页使用。

一般来说，直接**双击安装**就可以将这些字体嵌入你的计算机，使用的时候只要告诉计算机这个字体的名字即可。

本文使用的是Google的Noto字体系列，例如“Noto Sans CJK”，这个系列特别牛：
Noto的设计理念，就是“目标 No Tofu”，也就是**想让世界上每种语言的文字都能不出错地显示**。
Noto 字体跨越几乎所有已知语种，而且字型风格统一，非常适合跨国网站、国际软件或多语种内容。

py5提供了多种加载字体的方法，最常见的是用 `py5.create_font()` 和 `py5.text_font()` 实现。

- `py5.create_font(字体名, 大小)`：从系统或文件生成字体对象。

- `py5.text_font(字体对象)`：设置当前的文本绘制字体。
- 还可以用 `py5.text_size()` 设字号，用 `py5.text()` 显示文本。

有了字体文件，我们就可以愉快地在屏幕上绘制各种精美文字。
在屏幕上显示的文字，有很多属性可调，比如：

- **对齐方式（Alignment）：**
  - `py5.text_align(py5.CENTER, py5.CENTER)`：水平、垂直居中
  - `py5.text_align(py5.LEFT, py5.TOP)`：左上对齐
- **字体大小（Size）：** 用 `py5.text_size(大小)`
- **行间距、字间距、粗细、样式：**高级用法，还有字重/斜体选择
- **颜色/透明度：**和画形状一样，可以用 `py5.fill()` 控制
- 名词汇总
  - 字体（字形）：同一个文字的不同外貌、风格，比如宋体、楷体、Noto等。
  - 点阵字体：像积木一样用一排排像素点组成的字，一放大就会变模糊。
  - 矢量字体：用数学曲线描述字形，可以随意放大缩小都不会模糊。
  - 字体文件：储存一整套字体样子的电子文件，让计算机知道每个字怎么画。
  - 乱码（豆腐字）：电脑找不到字形时显示的方框或怪符号，看起来像一坨乱码豆腐。
  - Noto字体系列：谷歌出品、支持几乎所有语言、不掉豆腐块的全球通用字体系列。

In [15]:

```
import py5

def setup():
    py5.size(800, 520)
    py5.background(245)
    # 如果将这里注释掉，就会显示乱码
    # 因为py5默认字体不支持中文
    my_font = py5.create_font("Noto", 16)
    py5.text_font(my_font)
    py5.text_align(py5.LEFT, py5.TOP)

def draw():
    py5.background(245)

    # 绘制一行居中大字
    py5.text_size(42)
    py5.fill(16, 70, 170)
    py5.text_align(py5.CENTER, py5.TOP)
    py5.text("Hello, 世界! Py5字体演示", py5.width/2, 70)

    # 不同字号 + 颜色 对比
    py5.text_size(32)
    py5.fill(230, 60, 50)
    py5.text_align(py5.LEFT, py5.TOP)
    py5.text("小一号红色字体（左上角对齐）", 60, 160)

    py5.text_size(24)
    py5.fill(80, 190, 80)
    py5.text_align(py5.LEFT, py5.TOP)
    py5.text("更小号绿色，试试混合中文和English!", 60, 210)

    # 半透明&右对齐文字
    py5.text_size(30)
    py5.fill(120, 50, 200, 120)
    py5.text_align(py5.RIGHT, py5.TOP)
    py5.text("右对齐 / 半透明紫", py5.width - 60, 280)

    # 绘制示意区域分割线
    py5.stroke(150, 150, 150, 80)
    py5.stroke_weight(1)
    for y in [58, 150]:
        py5.line(30, y, py5.width-30, y)
    py5.no_stroke()

py5.run_sketch()
```

本章总结

本章知识点汇总

1. **像素与PPI**

- 像素（pixel）：屏幕、小图片或大画面上的“最小砖块”，每一个都是独立的小方格。
- PPI：像素密度，决定屏幕 / 图片“细腻度”。PPI越高，画面越细腻，单个像素点越看不清。

2. **坐标系**

- 数学坐标系：中心原点，x向右，y向上。
- 屏幕坐标系：左上角是(0,0)，x往右，y往下！这点超多新手会踩坑。
- 鼠标坐标系：实时获取鼠标在屏幕上的位置，做动态交互的基础。

3. **颜色**

- RGB与RGBA：屏幕发光靠红绿蓝三色混合，为颜色加“透明度”就是RGBA。
- 色域概念：不是所有设备都能显示所有颜色，高级显示器“地盘”更大。
- 反光系统（CMYK）和自发光系统（RGB）

- 反光系统（CMYK）和自发光系统（RGB）

4. **贝塞尔曲线**

- 用数学描述曲线路径。控制点越多，曲线越可控。

5. **文本与字体**

- 字体（字形）：同一个文字的不同外貌、风格，比如宋体、楷体、Noto等。
- 点阵字体：像积木一样用一排排像素点组成的字，一放大就会变模糊。
- 矢量字体：用数学曲线描述字形，可以随意放大缩小都不会模糊。

课后练习

1. 生活中有哪些地方可以看到“像素网格”或者“坐标纸”的影子？说说看你发现了哪些。
2. 请用通俗的比喻介绍什么是PPI，以及PPI高低对我们平时用手机、看电视的影响。
3. 假如Y坐标越大越“高”，交互设计可能会遇到哪些这样的问题？请举例。
4. 为什么在做动画或动态界面设计时，理解“相对坐标”和“世界坐标”很重要？试着联想到游戏或动画角色的运动。
5. 屏幕、鼠标、世界坐标这三种常见坐标各自适合什么场景？结合你的生活经验聊聊。
6. 用你自己的话解释RGB与CMYK分别适合“哪种世界”（屏幕还是印刷），用实例说明。
7. 试着列举3种你在平时生活中见到的“透明色”运用案例，并说说原理。
8. 回忆下“矢量图”和“位图”的差别：它们在交互设计里各有什么优势？
9. 请描述一下贝塞尔曲线在日常数字生活中的实际应用。可以举例一些不那么“工程”的例子。
10. 你觉得字体选择会对界面美观和交互体验造成哪些影响？谈谈你的审美看法。
11. 如果要给陌生人设计一套多语种的交互界面，如何利用字体选择兼顾美观与实用？
12. 设想做一个“颜色混合”互动程序，让用户体验不同的色彩组合。你会考虑哪些设计细节让用户觉得好玩？

扩展知识

无论使用位图还是矢量图保存图片，最终绘制在屏幕上时，都是通过点亮屏幕上的小方格——也就是**像素点**——来显示的。你可以想象屏幕像是一大片无数个微小“灯泡”拼成的墙，每个灯泡都能点亮不同的颜色，把我们看到的图片拼出来。所以不管最初是位图还是矢量图，最后都得“翻译”成由这些像素点组成的画面，这就是为什么在实际显示图片时，**本质上依然是位图**。

说到这里，有个小问题你可能会注意到——当我们用Py5画一条斜线或者曲线，如果你用放大镜（或把屏幕截图放大很多倍）去看，会发现曲线边缘有点像楼梯，呈现出一格一格的锯齿状，这就是常说的**锯齿现象**。为什么会这样？因为像素点本身就是一个一个小方块，组合起来难免会有“台阶”感觉。

那锯齿怎么办？这就派上了**抗锯齿（Antialiasing）**技术。可以把这个技术想象成用铅笔画线时，边缘地方轻轻地涂上一点淡颜色，让颜色慢慢过渡，这样曲线看起来就会更“丝滑”，没有那么多明显的台阶。实际上，抗锯齿就是让边缘的像素跟背景混合一部分颜色，让曲线变得更平滑，提升美观度。

除了画面细节，还有同学玩游戏的时候肯定见过“多重缓冲”、“双缓冲”、“垂直同步（VSync）”这些设置。为什么会有这些？简单来说，电脑画画不像我们用画笔那样，只要一动手就能画完一幅画。屏幕是不断刷新的，如果我们直接在显示的那一张画布上改动，很容易在画没完成的时候被屏幕刷新“偷看到”，这时候画面就可能出现“撕裂”，你可能会看到上半部分是“下一帧”，下半部分还是“上一帧”，像一张纸被撕了一样。

多重缓冲就类似于“备份画板”：

- **双缓冲**指的是准备两块画板（帧缓冲区）：一块负责当前显示，另一块在背后偷偷画，等画好以后两块互换。这样，观众（你）只会看到已经画完的完整画面，不会中途看到“半成品”，画面就不会撕裂。
- **三缓冲**则多加了一块画板，可以让图像切换和CPU、显卡之间的配合更灵活，减少等待，动作更流畅。

再说说**垂直同步（VSync）**，它其实就是让画面的刷新和屏幕的实际刷新节奏对齐——只有在屏幕准备好下一次全部刷新时，才允许画板的内容换上来，这样能进一步防止撕裂。

这些技术，和我们用Py5画动图、做交互的时候，其实是一个道理！比如你写个动画小球，Py5背后就悄悄用着双缓冲技术，这样你看到的小球滚动才会平滑自然。

如果你对这些底层原理感兴趣，推荐可以去网上搜“计算机图形学入门”，“游戏渲染流程”，或者参考Py5官方文档的动画和缓冲相关章节。
也可以找一些讲显卡和屏幕原理的科普视频看看，绝对会有新发现！

最早的“屏幕显示”，你可能想不到，其实是一些简单的二极管和小灯泡。比如早期的计算机（比如ENIAC那种庞然大物），它们在面板上布满了小灯泡——某个电路通了电，那个灯泡就亮。这不能算是真正意义上的“屏幕”，但已经可以把数字、状态用“灯”的亮灭来表达，是最朴素的“显示技术”雏形。

到了20世纪中期，出现了著名的**CRT（阴极射线管）**。你家里如果有爷爷奶奶的老电视，还能看到就是那种又大又厚又沉的“老电视”，俗称“大屁股电视”。它的原理其实很酷，就是用电子枪朝荧光屏发射电子束，打到哪里哪里就亮，颜色和亮度

的“老电视”，俗称“大屁股电视”。它的原理其实很简单，就是用电子枪朝荧光屏发射电子束，打到哪里哪里就亮，颜色和亮度靠不同的电子束调整。这就是我们最早的“像素”屏幕，从此电视、计算机“看起来有内容”这件事才真正流行开。

再后来，如果你1980、90后，电脑或电视就流行起了“扁平”的**LCD（液晶显示屏）**。液晶分子排成一行一列，利用液晶材料控制光线的透过与阻挡来显示图像。

真正革命的一步是什么？就是**OLED（有机发光二极管）**！它终于做到了——每个像素都能自己发光（不用背光板了），这样什么好处呢？

- **对比度爆表**：黑的地方彻底不亮，真的黑；
- **颜色超鲜艳**：因为每颗像素独立控制，颜色层次还原更棒；
- **更薄更柔软**：OLED可以做成弯曲甚至折叠的形态，所以现在有了可折叠手机、卷轴电视这些黑科技。

你可以这样理解屏幕进化路线：

- 从一串小灯泡（只能表达亮 or 灭），
- 到大屁股电视（CRT，全屏可显示基础像素图像），
- 到LCD（让屏幕更薄，可便携），
- 最后到OLED（每个像素自己“发光”，对比度、鲜艳度都爆表，形态任意变化）。

虽然OLED可以从各方面表现出极其鲜艳、饱和甚至让人惊艳的颜色，但是作为交互设计师，你需要明白：色彩不仅仅是用来“好看”的，更是用来**传达信息、引导注意力、提升可用性和舒适度**的“语言”工具。

比如说，如果你把所有界面都设计得色彩丰富、对比极高，确实乍一看很“炫酷”，但用户可能会被强烈的亮色刺激得眼睛不舒服，长时间使用甚至会造成视觉疲劳。再比如，OLED自发光的黑色是真的“黑”，暗色模式不光省电、屏幕寿命长，还能降低夜间用眼压力——但如果用色不对，可能导致信息辨识度下降，用户在强光下（比如户外看手机）反而看不清楚。

此外，色彩在交互设计里还要承载很多“功能性”角色，比如用红色快速提醒用户哪里出现错误、绿色提醒一切顺利，或者用特别的品牌色来加强界面风格和用户认知。不只是美观，要让色彩成为沟通和引导的帮手。所以你还需要思考色彩的可读性、色差对色盲的影响、不同文化背景下的色彩含义等等。

同时，OLED的高对比和高饱和也可能把原本设计好的界面色彩表现“推过头”——有些颜色会比你在普通屏幕或电脑显示器上看到的更刺眼、更鲜明，甚至改变了本来的风格。这意味着你可能要“为OLED适配”做些细致调整，比如校准色值、避免长时间显示高亮固定元素防止烧屏、多设计暗色主题、提高动态适应环境能力，让用户始终感觉界面是“为他们而设计”而不是强行炫技。

- **苹果的 Human Interface Guidelines（HIG）** 里专门有色彩使用的章节，不仅讲怎么搭配才美观，还特别强调了怎么让内容清晰、可读、适合不同环境，还有针对 OLED/暗色模式的细节建议。你可以阅读这里：
  Human Interface Guidelines: Color

- **安卓的 Material Design 指南**也有一套体系化的色彩系统，里面结合了心理学、可访问性、品牌感知等实用建议，讲述了怎样让色彩既贴合产品逻辑、又适合各种类型设备屏幕。推荐这里：
  Material Design: Color System

这些指南不仅适合专业设计师，也非常适合我们做交互编程和界面设计的同学参考。边做边学，你会发现即使是在屏幕不同、硬件不同的情况下，也能做到色彩既美观又实用，真正服务于用户体验。

练习题提示

1. 想想自己的美术课、手机屏幕、拍照App里，哪类工具其实本质用到了像素或坐标思想？
2. 用“马赛克照片”或“拼豆娃娃”来联想高/低PPI画面的效果差别，体验一下不同清晰度的感受——可以去截屏/放大对比。
3. 屏幕坐标的Y轴朝下，想象角色从屏幕顶部“掉落”下来，坐标变化会如何？什么情况下容易出现理解偏差或BUG?
4. 游戏、动画很多角色/物体其实相对自己的小世界运动，和大世界有映射。用切换地图、人物换装等例子帮助理解。
5. RGB适合给屏幕颜色调色，CMYK更多是给打出来的杂志调色，请集思广益、观察身边物品区别。
6. “半透明色”常见于手机通知栏、弹窗遮罩、PPT高亮区域等交互。你还想到哪些？能否解释为什么要“半透明”而不是“纯色”?
7. 位图放大“糊了”，矢量图放大还很清晰。可以试着用手机拍一下书上的印刷字体和电脑上的同一文字做对比。
8. 贝塞尔曲线上滑动的“小球”与你生活中的哪些线条或动画相似？比如地铁地图线、画板里画的爱心、手机里的涂鸦等。
9. 字体选择哪里看得见最直接的影响？比如浏览新闻网站、打开不同品牌手机的短信App等等，用截图、观察比较收获会很大。
10. 多语种界面场景，去看看常见的跨国网站（比如Google、Twitter）如何用字体兼容各种语言，还特别好看。

游戏循环：初始化、主循环（事件更新、运动更新、界面更新）、结束

Processing：setup与draw

鼠标信息：click、position

In [ ]:

```
import py5

red = True
mouseSpeed = 1

def setup():
    py5.size(500, 500)
    py5.stroke(255, 0, 0)

def draw():
    mouseSpeed = abs(py5.mouse_x - py5.pmouse_x)
    py5.stroke_weight(mouseSpeed)
    py5.line(py5.pmouse_x, py5.pmouse_y, py5.mouse_x, py5.mouse_y)

def mouse_pressed():
    py5.stroke(0 if red else 255, 255 if red else 0, 0)

py5.run_sketch()
```

第3章练习：根据用户按键改变画笔颜色