



DDLC Mod

开发指南

DDLC MOD Development Guide

(Ver. 0.1.0-alpha)

PartyParrot 著

DDLC Mod Development Guide © 2023 - 2025 by PartyParrot is licensed under [CC BY-NC-SA 4.0](#).
This work is based on *The Legrand Orange Book* Version 3.1 by Vel and Mathias Legrand under [CC BY-NC-SA 4.0](#) and *Doki Dokiliterature Club!* by Team Salvato under IP Guidelines.
PartyParrot 所著的 DDLC Mod 开发指南 © 2023 - 2025 基于[CC BY-NC-SA 4.0](#)分发。
本作品在[CC BY-NC-SA 4.0](#)下使用了 Vel 和 Mathias Legrand 编写的 *The Legrand Orange Book* Version 3.1, 在 IP Guidelines 下基于 Team Salvato 的 *Doki Dokiliterature Club!*。



前言

Ren'Py 是一款开放源代码的自由软件引擎，用来创作透过电脑叙述故事的视觉小说。Ren'Py 之名是 Ren'ai 与 Python 两词混合而成。Ren'ai 为日文，意指“恋爱”，而 Python 是 Ren'Py 所使用的语言环境。

《心跳！心跳！文学部！》(*Doki Doki Literature Club!*, 下文简称 DDLC) 是一款基于 Ren'Py 编写的游戏。因此，要编写 DDLC 的 Mod 就必须学习 Ren'Py。

由于 Ren'Py 与 Python 密不可分，本书在介绍 Ren'Py 特性的同时，还讨论了基本 Python 用法，使二者称为有机的整体。书中介绍了 Ren'Py 的基本概念，通过简单明了的例子向读者展示 Ren'Py 的代码与知识。

本书共两部分，分为基础部分与进阶部分，分别介绍了 Ren'Py 的对话系统、图像显示系统等内容。

本书针对 Ren'Py 的初学者，从 Ren'Py 基础知识开始介绍，并在教学过程中会介绍部分 Python 知识，因此不要求读者有 Ren'Py 和 Python 的基础知识，本书中部分内容对于部分读者可能会需要一点的基础的计算机知识，请善用百度。

初级教程方法

在 *C++ Primer Plus* 的前言中，有这么一段话：

大约 20 年前，*C Primer Plus* 开创了优良的初级教学传统，本书建立在这样的基础之上，吸收了其中很多成功的理念。

- 初级教程应当是友好的、便于使用的指南。
- 初级教程不要求您已经熟悉相关的编程概念。
- 初级教程强调的是动手学习，通过简短、容易输入的实例阐述一两个概念。
- 初级教程用示意图来解释概念。
- 初级教程提供问题和联系来检验您对知识的理解，从而适于自学。

...

本书的作者和编辑尽最大的努力使本书简单、明了、生动有趣。我们的目标是，您阅读完本书后，能够编写出可靠、高效的程序，并且觉得这是一种享受。

本书秉持着与 *C++ Primer Plus*, *C Primer Plus* 同样的理念，通过简单明了的例子、逻辑连贯的项目式的代码来使读者理解、明白 Ren'Py 的代码与特性。

示例代码

本书包含大量的示例代码，其中大部分代码是一段完整的、可独立运行的代码。由于 Ren'Py 的特殊性，大部分代码需要放在相应环境中才能运行。本书代码只适用于 Ren'Py 7/8，理论上可以在 Ren'Py 6 上运行，但由于其过时性，本书将不再针对 Ren'Py 6 进行介绍与适配。

本书示例代码及注释样式

为区别普通文本，本书对于实例代码做出以下规定：

- 代码使用 14 点的 Sarasa Term SC 字体。背景为 RGB 颜色 (235, 235, 235)。如：

```
# 这是一行注释
```

- 需要您输入的内容将以粗体出现。如：

```
1 $ renpy.input()
2 # 输入: 22
```

- 表示代码输出结果的将以斜体出现。如：

```
1 >>> 1 + 2
2 # 输出: 3
```

- 语法中的占位符将用尖括号括起来。您应使用实际的参数、变量等替换占位符。如：

```
define <变量名称> = <值：整型、浮点型等>
```

您应将其替换类似的例子：

```
define a = 2
```

- 当代码中不含有 »> 或... 则表示您应该使用文件运行代码，而非 Python 交互模式。
- 本书中只能在 Python 代码块中运行的语法，将会含有 **[Python Only]** 标签。如下例：

for 循环 [Python Only]

try-except [Python Only]

同时，本书分为 4 种注释类型：

- 普通注释背景使用 25% 色调青色，边框使用 75% 青色，如：

注释

这是一行注释。

- 扩展知识背景使用 25% 色调绿色，边框使用 RGB 颜色 (105, 190, 78) 如：

扩展知识

这是一行扩展知识。

- 警告背景使用 25% 色调黄色，边框使用 RGB 颜色 (150, 150, 0) 如：

警告

这是一行警告。

- 必须注意的内容背景使用 25% 色调红色，边框使用 75% 红色，如：

必须注意

您必须注意此内容。

开发本书编程示例所使用的系统

本书的 Ren'Py 示例是使用 Ren'Py SDK 7.6.1 开发的，在 Ren'Py SDK 7.8.1、Ren'Py 8.0.3、Ren'Py 8.3.7 上进行过测试，同时代码在 Arch Linux、Windows 11 24H2 64 bit 上的进行了测试。

获取最新版指南

<https://wwyc.lanzouq.com/b02fb2saj>

密码:ddlc

<https://github.com/DanilJeston/DDLC-Chinese-Modding-Guide>

联系方式

我们的联系邮箱是: team_ninety@outlook.com。

如果您对本书有任何疑问或建议，请发邮件给我们。若您有兴趣参与本书的编写、完善，可以邮件给我们。同时，若您发现有人未经 CC BY-NC-SA 4.0 方式分发本书，请发邮件给我们。若本书存在部分代码出现错误、无法运行等，请发送邮件给我们。

最后，祝您学习愉快！



目录

前言

i

第一部分 基础部分

1

第一章 预备知识

2

1.1 关于 Ren'Py	2
1.1.1 Ren'Py 概述	2
1.2 下载 Ren'Py	2
1.2.1 下载 Ren'Py SDK 7	3

第二章 开始学习 Ren'Py

5

2.1 准备开发环境	5
2.1.1 下载 DDLC Mod 中文模版	6
2.1.2 配置文本编辑器	6
2.2 进入 Ren'Py 世界	7
2.2.1 say 语句	8
2.2.2 show、hide、scene 语句	11
2.2.2.1 show 语句	11
2.2.2.2 hide 语句	14
2.2.2.3 scene 语句	14
2.2.3 play、stop 语句	15
2.2.3.1 play 语句	15
2.2.3.2 stop 语句	16
2.2.4 menu 语句	16

第三章 Python 与 Ren'Py

18

3.1 Python 中的几种数据类型	19
-------------------------------	----

3.1.1 数字	19
3.1.2 字符串	20
3.1.3 列表与元组	21
3.1.3.1 列表	21
3.1.3.2 元组	23
3.1.4 字典	23
3.1.5 布尔类型	24
3.1.6 小节	25
3.2 函数与类 [Python Only]	26
3.2.1 函数	26
3.2.1.1 函数的定义	26
3.2.1.2 函数命名空间	27
3.2.2 类	29
3.2.2.1 何时使用类?	29
3.2.2.2 类的定义	31
3.2.2.3 类的使用	31
3.3 在 Ren'Py 中使用 Python 语句	32
3.3.1 Python 语句块	32
3.3.2 单行 Python 语句	33
3.3.3 init python 语句	34
3.4 使用变量	35
3.4.1 python 语句块	35
3.4.2 define 语句	35
3.4.3 default 语句	36
3.4.4 持久化数据	36
3.5 流程控制	36
3.5.1 脚本标签	36
3.5.1.1 label 语句	36
3.5.1.2 call 语句与 jump 语句	37
3.5.2 if 判断	39
3.5.3 循环	40
3.5.3.1 while 循环	40
3.5.3.2 for 循环 [Python Only]	41
3.5.4 错误和异常 [Python Only]	44
3.6 等效语句	46
3.6.1 对话	46
3.6.2 图像显示	46
3.6.2.1 show	46
3.6.2.2 hide	46
3.6.2.3 scene	46
3.6.2.4 with 从句	46
3.6.3 call 和 jump	46

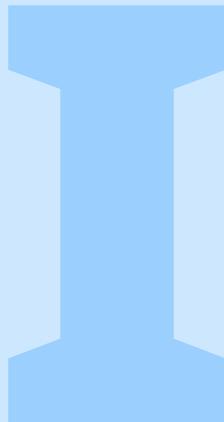
第四章 增添资源	48
4.1 增添资源	48
4.1.1 定义角色	49
4.1.2 增加、定义图片	50
4.1.2.1 角色立绘	50
4.1.2.2 背景图像	51
4.1.3 音频资源	52
4.1.3.1 节选播放	52
第五章 特殊效果及特殊脚本	53
5.1 特殊脚本	53
5.1.1 bsod.rpy	53
5.1.2 cgs.rpy	53
5.1.3 console.rpy	53
5.1.4 glitchtext.rpy	54
5.1.5 poems_special.rpy\poems-tl.rpy\poems.rpy	54
5.2 特殊效果	54
5.2.1 撕裂效果	54
5.2.2 死机效果	55
5.2.3 对话修改效果	55
5.2.4 画面心跳效果	56
5.2.5 操控历史对话	56
第六章 分发 Mod	57
6.1 修改设置	57
6.2 打包模组	58
第二部分 进阶部分	60
第七章 更新 Ren'Py	61
7.1 判断您是否需要升级	61
7.2 更新 Ren'Py 版本与模板版本	62
7.2.1 更新 Ren'Py	62
7.2.2 更新模板	62
7.2.3 解除兼容性警告	63
第八章 自定义 GUI 与定义新界面	64
8.1 Ren'Py 的界面显示机制	64
8.2 自定义 GUI	65
8.3 界面语言	65
8.3.1 初步了解界面语言的结构	65



表格列表

2.1 角色名表格	8
2.2 常用文本处理标签	9
2.3 常用的 show 语句附加参数	12
2.4 角色状态表	13
2.5 常用转场动画编号及其效果	13
3.1 常用逻辑运算符号与含义	20
3.2 常见逻辑、比较运算符与示意	24

基础部分



第一章 预备知识	2
1.1 关于 Ren'Py	2
1.2 下载 Ren'Py	2
第二章 开始学习 Ren'Py	5
2.1 准备开发环境	5
2.2 进入 Ren'Py 世界	7
第三章 Python 与 Ren'Py	18
3.1 Python 中的几种数据类型	19
3.2 函数与类 [Python Only]	26
3.3 在 Ren'Py 中使用 Python 语句	32
3.4 使用变量	35
3.5 流程控制	36
3.6 等效语句	46
第四章 增添资源	48
4.1 增添资源	48
第五章 特殊效果及特殊脚本	53
5.1 特殊脚本	53
5.2 特殊效果	54
第六章 分发 Mod	57
6.1 修改设置	57
6.2 打包模组	58



1. 预备知识

Ren'Py 是一种视觉小说语言，名字是恋愛（れんあい，即恋爱）与 Python 两词混合而成。Python 是 Ren'Py 使用的编程语言。而《心跳！心跳！文学部！》(Doki Doki Literature Club!, 下文简称 DDLC) 正是基于 Ren'Py 编写的。想要编写 DDLC Mod，我们就必须先学习 Ren'Py 相关知识。

本章将为您介绍 Ren'Py 的来源及如何安装 Ren'Py 与代码编辑器。

1.1 关于 Ren'Py

1.1.1 Ren'Py 概述

Ren'Py 几乎支持视觉小说所应该具有的功能，如：分支故事、存储与加载游戏、回退到之前故事的存储点、多样性的场景转换等。其首次发布于 2004 年 8 月 24 日。Ren'Py 拥有类似电影剧本的语法，并且能够允许用户编写 Python 代码来增加新的功能。除此之外，游戏引擎内附的出版工具能提供基本的脚本加密与压缩游戏素材。

Ren'Py 建构于 Python 软件库 Pygame 之上，而它又基于了 SDL。Ren'Py 官方支持 Windows、Linux 以及较新版 Mac OS X，并可通过 Arch Linux、Ubuntu、Debian 或 Gentoo 的软件包管理系统安装。它可以在 Windows、macOS、Linux、Android、OpenBSD、iOS 和 wasm 的 HTML5 下建置。

利用 Ren'Py 结合剧本及 Python，我们可以制作出各种各样的游戏。Ren'Py 也有一些电子角色扮演游戏框架的示例，但相对来说，制作 RPG 游戏会比较困难。

1.2 下载 Ren'Py

目前主流的 Ren'Py 分为 3 个版本：Ren'Py 6、Ren'Py 7、Ren'Py 8。其中 Ren'Py 6 与 Ren'Py 7 兼容 Python 2 且 Ren'Py 7 已经支持大部分 Python 3 特性，而 Ren'Py 8 完全兼容 Python 3。需要注意的是，Ren'Py 6 已停止支持。

Ren'Py 几乎可在所有主流系统上运行。由于 Ren'Py 8 目前稳定性存疑，故本书主要使用 Ren'Py 7 进行教学。

注释

Ren'Py 7.4.4 及以后的版本均不支持 Windows XP 及更早的系统。

1.2.1 下载 Ren'Py SDK 7

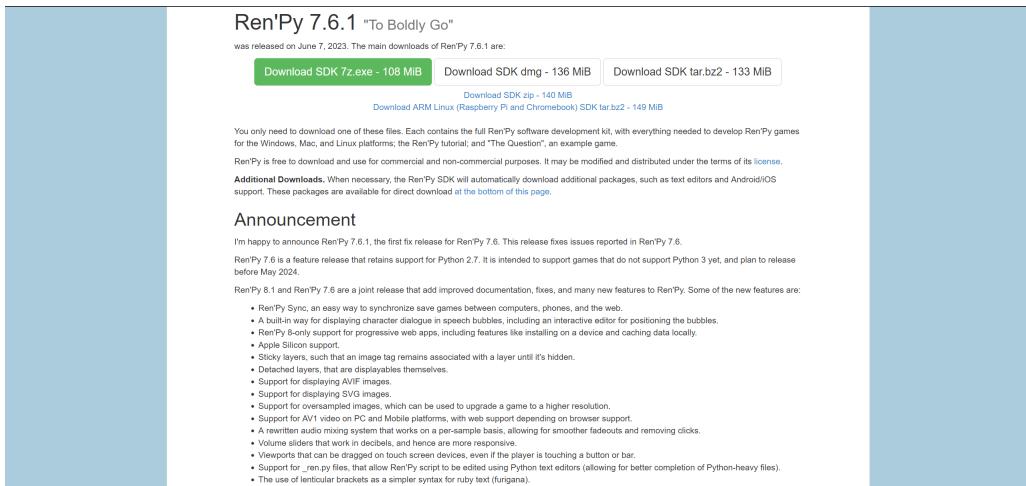


图 1.1: Ren'Py 官网

注释

本章编写时 DDLC Mod 中文模板支持的最新版 Ren'Py SDK 7 为 7.6.1。

使用任意浏览器进入 <https://www.renpy.org/release/7.6.1> 网页 (如图1.1)，您将在本页面看到三个按钮，分别是：Download SDK 7z.exe、Download SDK dmg 与 Download SDK tar.bz2。Windows 用户请下载第一个，macOS 用户请下载第二个，Linux 用户请下载第三个。

注释

若无法打开文件，请点击第二行小字 Download SDK zip 下载 ZIP 文件



图 1.2: ZIP 文件

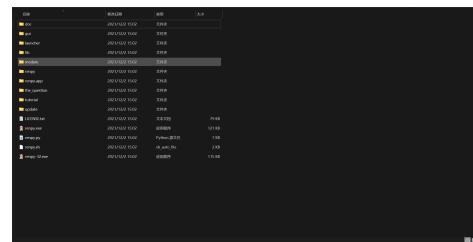


图 1.3: 文件夹内容

以 ZIP 文件为例，下载完成后打开压缩包（如图1.2）将文件夹中的所有内容解压，得到如下内容。（如图1.3）

Windows 用户请双击 renpy.exe 或 renpy-32.exe(在某些情况下, 它可能显示为 renpy 或 renpy-32); MacOS 用户若使用驱动器镜像 (dmg) 安装 Ren'Py, 挂载后将 renpy-7.6.1 复制到桌面, 进入桌面上的文件夹并运行 renpy。; Linux 用户请运行 renpy.sh。

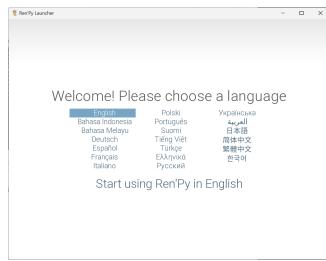


图 1.4: 选择语言

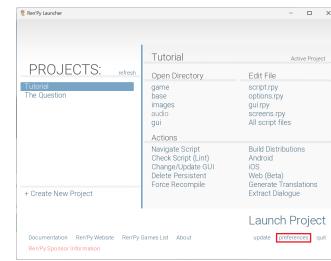


图 1.5: 调整语言

打开 Ren'Py 后, 正常来讲, 您会见到如图1.4所示的界面。点击简体中文后, 您会见到如图1.5所示的界面。

若您没有见到该界面, 而是直接见到类似图1.5, 请点击右下角的 preferences, 在右下角 Language 中选中简体中文即可调整为简体中文。



2. 开始学习 Ren'Py

本节目标包括：

- 使用 DDLC Mod 中文模版创造一个项目；
- 学会使用 say 语句；
- 学会使用 show、hide、scene 语句；
- 学会使用 play、stop 语句；
- 学会使用 menu 语句；

要建造一个属于我们的房屋，就必须要先打地基、搭框架。如果说 Ren'Py 是项目的地基，那么 Mod 模板就是一个框架，我们可以在这个框架中快速地放置门、窗等，还可以自定义这个框架，而不必先调制水泥，然后从地基一步步开始。通过模版，我们可以省去很多步骤。本章我们将在 DDLC Mod 中文模版的基础上将对 Ren'Py 进行初步学习。

2.1 准备开发环境

DDLC Mod 模板是由 GanstaKingofSA 编写的一个方便开发 Mod 的模板。imgradeone 对其进行了本土化。目前，DDLC Mod 中文模板主要流行三个大版本：1.0 版本，2.0 版本（即 Next 分支）、4.0 版本（即 Future 分支）、与 5.0 版本。

- 1.0 版本仅支持 Ren'Py 6，且没有什么特别功能；
- 2.0 版本增加支持了 Ren'Py 7、Android 移植等功能；
- 4.0 版本增加支持了 Ren'Py 8 与 Python 3，增加额外屏幕 (Extra Screen) 功能等，但稳定性存疑；
- 5.0 版本在 4.0 的版本上全面舍弃了对 Ren'Py 7 与 Python 2 的支持，GUI 界面自定义功能更强，但稳定性存疑。

本书该部分将使用 2.0 版本进行讲解。

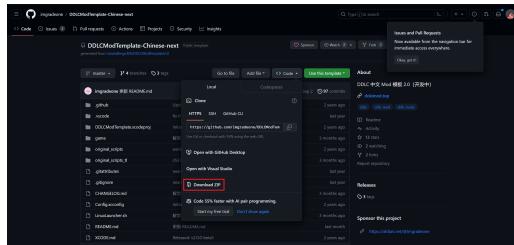


图 2.1: 项目页面

2.1.1 下载 DDLC Mod 中文模版

使用任意浏览器打开<https://github.com/imgradeone/DDLCModTemplate-Chinese-next>, 将压缩包下载下来(如图2.1所示)。

下载完成后解压至在第1.2.1节中您下载的 Ren'Py SDK 的目录下。完成该步骤后，您的 Ren'Py SDK 中项目一栏应出现 DDLCModTemplate-Chinese-next。

从 <https://ddlc.moe/> 中下载原版 DDLC, 打开压缩包后将 DDLC-1.1.1-pc/game/下的 audio.rpa、images.rpa、fonts.rpa 解压至 Mod 中文模板下的 game/文件夹中。此时，Mod 中文模板下的 game/文件夹结构应如图2.2所示。

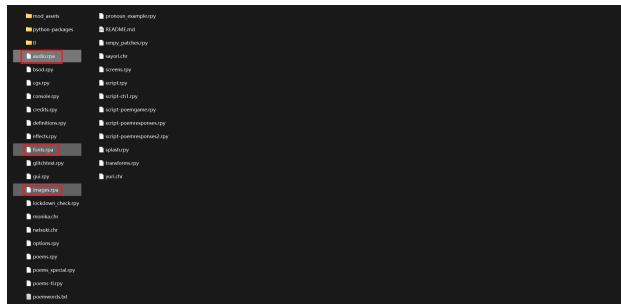


图 2.2: 项目结构

至此，您就完成了准备工作的第一部分——下载 DDLC Mod 中文模板并完成配置。

2.1.2 配置文本编辑器

有了 Mod 中文模板，我们还需要一个趁手的编辑器。你大可选择记事本，不过 Ren'Py SDK 也为我们提供了两种文本编辑器:Visual Studio Code 和 Atom。本书将以 Visual Studio Code 为例配置文本编辑器。

打开 Ren'Py SDK, 点击右下角的设置。在一般选项卡中选择文本编辑器，此时点击第一个选项 Visual Studio Code。Ren'Py 会开始下载 Visual Studio Code 与 Ren'Py 插件。耐心等待一段时间后，会返回到设置界面。此时点击返回，点击 DDLCModTemplate-Chinese-next，点击编辑选项卡下的“打开项目”会打开 Visual Studio Code(如图2.3所示)。

此时点击扩展选项卡，输入 Chinese，点击第一个搜索结果中的 Install。此时右下角会弹出提示框询问是否切换语言并重启。点击 Change Language and Restart。重启后 Visual Studio Code 就会切换成中文。

打开 Ren'Py SDK, 选中 DDLCModTemplate-Chinese-next，点击编辑文件选项卡中的“打开项目”，Ren'Py SDK 会自动帮我们打开 Visual Studio Code 并定位到本项目。

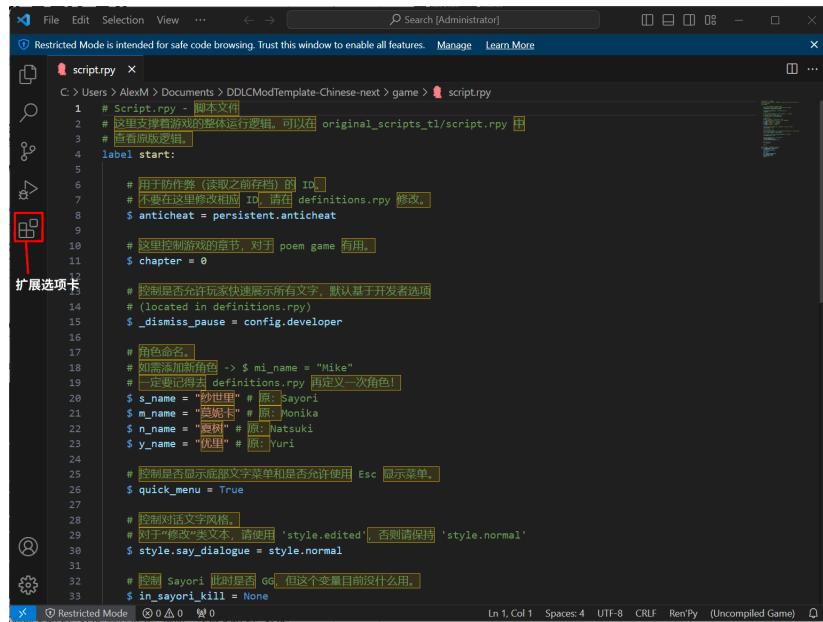


图 2.3: Visual Studio Code

随后，展开左侧资源管理器中的“game”文件夹，打开 script-ch1.rpy 文件。保留文件第一行，删除其他内容即可。至此，我们就正式完成了开发的准备工作。

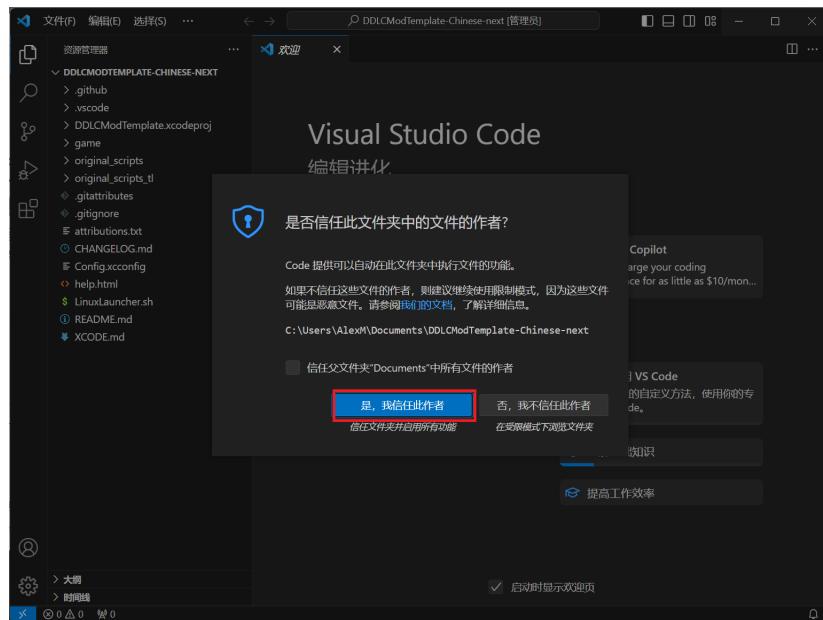


图 2.4: Visual Studio Code

若您打开 Visual Studio Code 时出现如图2.4所示的提示框，请点击“是，我信任此作者”。

2.2 进入 Ren'Py 世界

在 Ren'Py 的世界里，剧本与视觉内容都围绕着代码展开。要编写脚本，我们就必须先学习 Ren'Py 的语法。

2.2.1 say 语句

say 语句是一个十分重要的语法，它可以让视觉小说角色拥有对话的能力。不过不用担心，say 语句的语法十分简单。

还记得上一课我们打开的 script-ch1.rpy 吗？现在另起一行，打四个空格或按下 Tab 键（Visual Studio Code 会自动为我们把 Tab 转化为空格），输入以下代码片段：

```
1 "快到学园祭了"
2 m "各位！ 我们得开始准备了！"
```

警告

您必须使用英文引号来包住文字。另外，请务必注意缩进问题，错误的缩进将会导致代码运行失败。

这是一段非常简单的 Ren'Py 代码，它只包含文字，不显示任何图像或播放任何音效。现在，切换回 Ren'Py SDK，点击启动项目，开始游戏，您应该会看到一行旁白以及莫妮卡的一行对话。

在理解以上基础内容之后，我们可以添加更多对话。不过在那之前，你得先了解 say 语句的基础语法：

```
<角色名> "<说话内容>"
```

此处的角色名应对应下列表格：

表 2.1: 角色名表格

角色名	对应角色
m	莫妮卡 (Monika)
s	纱世里 (Sayori)
y	优里 (Yuri)
n	夏树 (Natsuki)
mc	主人公 (Main Character)
留空	旁白 (Narrator)

现在，尝试理解下列代码并运行：

```
1 # Chapter 0
2 label ch0_start:
3     "快到学园祭了。"
4     m "各位！ 我们得开始准备了！"
5     s "好耶！！！！"
6     n "啊， 我都等不及学园祭了。"
7     n "肯定会很棒的！"
8     y "... "
9     return
```

代码 2.1: script-ch1.rpy

在 Ren'Py 中，我们使用脚本标签（label）来声明代码块。（关于脚本标签的知识，我们将会在后续内容进行详细讲解）。通常，在 Ren'Py 项目中都含有一个名为 start 的脚本标签。

但是在 DDLC Mod 模板中，script.rpy 文件已经为我们声明好了 start 脚本标签。在本段代码中，script-ch1.rpy 包含以下几件事：

- 第一行的注释；
- 创造一个名为 ch0_start 的代码块；
- say 语句创造对话；
- return 语句返回到上级。

注释

在 Ren'Py 中，我们使用与 Python 相同的“#”来表示本行为注释。注释行的内容将不会被 Ren'Py 或 Python 执行。

当我们需要在对话中提及玩家的名字、需要将某些内容突出显示或对某部分文字需要进行特殊处理时，应当怎么办呢？读代码2.2，尝试理解其用法，并在游戏中运行，看看与自己的猜测是否相同。

```

1 # Chapter 0
2 label ch0_start:
3     "{cps=20} 快到 {b} 学园祭 {/b} 了。{/cps}{w=.5}{nw}"
4     m "各位！ 我们得开始准备了！"
5     s "好耶！！！！"
6     n "啊， 我都等不及学园祭了。"
7     n "肯定会很棒的！"
8     y "... "
9     m "那么， 是时候来进行分工了。"
10    m "[player]， 你想要做什么？"
11    return

```

代码 2.2: script-ch1.rpy

扩展知识

player 是一个记录了玩家名字的变量。不理解变量是什么？没关系，本书将在后面为您介绍变量的性质与作用。在这里，您只需要知道当需要提及玩家名字时使用 **player** 变量即可。

扩展知识

您或许也注意到了 **{/cps}** 和 **{/b}**。这是对前面 **cps** 标签和 **bold** 标签的闭合，代表着其适用范围仅为自身标签与闭合标签内的所有文字。

不难发现，当需要使用变量时，我们只需要使用中括号将变量名括住。当我们需要将某段文字进行加粗等操作，可以对文字打上标签（既花括号）。若需要在对话中使用这些字符，则需要连续出现两次。如在对话中要出现花括号，则应该使用 **{()}** 代替 **{}**

以下为常用的文字标签：

表 2.2: 常用文本处理标签

标签	作用
{b}	将文本渲染为粗体
{i}	将文本渲染为斜体
{color=<16进制颜色>}	将文本渲染为指定颜色
{font=<字体文件>}	使用指定字体渲染文本
{cps=<数字>}	以指定速度渲染文本
{nw}	不等待用户操作，渲染完本行文字后立即渲染下一行
{w=<数字>}	等待一定秒数后或用户操作后继续渲染文本

2.2.2 show、hide、scene 语句

在2.2.1课中，我们学习了如何在视觉小说中编写对话。但现在有一个很大的问题——没有图像。在视觉小说中，视觉便是重点。在本课中，我们将学习如何在视角小说中显示图像。

2.2.2.1 show 语句

show 语句是 Ren'Py 游戏中的一个重要语句。它可以让一个图像显示在屏幕上。这个图像可以是背景，也可以是角色。

还记得上一课中的代码吗？请阅读以下代码并尝试理解，在 Ren'Py 中运行，看看运行实际效果与自己的理解是否正确。

```

1 # Chapter 0
2 label ch0_start:
3     show bg club_day
4     "{cps=20}快到{b}学园祭{/b}了。{/cps}{w=.5}{nw}"
5     show monika 1a at t41 zorder 1
6     m "各位！ 我们得开始准备了！"
7     show sayori 1a at t42 zorder 1
8     s "好耶！ ！！！"
9     show natsuki 1a at t43 zorder 1
10    n "啊， 我都等不及学园祭了。"
11    n "肯定会很棒的！"
12    show yuri 1a at t44 zorder 1
13    y "... "
14    m "那么， 是时候来进行分工了。"
15    m "[player]， 你想要做什么？"
16    return

```

代码 2.3: script-ch1.rpy

由此可知，show 语句的基础语法为：

`show <图片名称> <附加参数>`

在游戏中，图片名称可以大致分为三类：

- 角色立绘
- 背景立绘
- 毛刺 (Glitch) 立绘

角色立绘 对于角色立绘，图片名称的组成为：

<角色名> <立绘编号>

DDLC 原版中，一个显示在屏幕上的角色立绘可以分为两个部分：身体姿势与面部表情。（实际上，DDLC 中的立绘为三个部分：左侧身体姿势、右侧身体姿势、面部表情）

模版中已经将所有可能的身体姿势为我们组合好了。除优里外的所有角色都有五个姿势（1-4 为正常站位，5 为特殊站位），而优里只有 4 个姿势（1-3 为正常站位，4 为特殊站位）。

面部表情使用英文小写字母来表示（请注意，不是所有的角色的面部表情都表示到 z。如莫妮卡的面部表情就远少于夏树的面部表情）。

同时，除莫妮卡外的所有角色都可以使用日常服，只需要在身体姿势后加上 b 即可使用日常服。如：1b

对于角色立绘的列举，您可以前往 <https://docs.dokimod.top/pages/0a59cf/> 与 https://ddlc-modding.fandom.com/wiki/Expressions_and_Poses 查看。

毛刺立绘 毛刺立绘则是指在二周目出现的大多数“错误”立绘，如错乱的莫妮卡等。对于毛刺立绘，其图片名称的构成为：

<角色名> <毛刺编号>

警告

我们不建议您使用毛刺立绘，因为这对于某些厨可能极不友好。

一般来说，毛刺编号为 glitch。但是请注意，莫妮卡的毛刺立绘编号为 g1 与 g2；夏树的毛刺立绘则包括 ghost_blood、ghost1、ghost2 等。

若要使用优里的自杀立绘，则毛刺编号为 stab_+ 数字 1-6。

背景立绘 对于背景立绘，其图片名称构成为：

1 bg <背景编号>

背景编号即意义可以在 definitions.rpy 文件中找到。这里不再进行过多叙述。

扩展知识

通常我们不使用 show 来显示背景，而是使用 scene。对于 scene 的详细介绍请看第 2.2.2.3 课。

附加参数 对于附加参数，常用的有：

表 2.3: 常用的 show 语句附加参数

语法	作用
at <transform>	将一个动画 (transform) 套用到立绘上
with <transform>	显示图像时使用指定的转场动画 (transform)
zordr <数字>	控制图像在 Z 轴上的位置

扩展知识

show 语句也可以使用 behind 从句、as 从句、onlayer 从句。但由于在游戏中这些从句很少会使用，故不再展开讲解。有兴趣者可前往 <https://doc.renpy.cn/zh-CN/dis>

[playing_images.html](#) 了解详细内容。

at、with 从句 变换 (transform)，就如名字一样，定义了一个图像在屏幕上的变化内容与方式，即动画、运动方式、位置、透明度等。当一个变换作用于角色立绘时，应使用 at 从句。一个例子：t41。在这个变换中，t 表示在这个位置上的角色静止站立在原地；4 表示一共有 4 个站位，会将屏幕等分成 4 份；1 表示从左往右该变换是所有站位中的第 1 个。

所有适用于角色立绘的变换有：

表 2.4: 角色状态表

编号	意义
t	角色静止站立在原地
i (instant)	角色突然出现
f (focus)	角色成为屏幕焦点
s (sink)	角色下沉
h (hop)	角色跳跃
hf (hop and focus)	角色跳跃并成为屏幕焦点
d (dip)	角色向下倾斜然后升起
l (left)	角色从左侧飞入
r (right)	角色从右侧飞入

对于站位总数，共有 1 到 4。

当变换作为转场动画来使用时，应使用 with 从句。游戏中实现定义好的转场动画有：

表 2.5: 常用转场动画编号及其效果

编号	意义
dissolve	溶解效果
dissolve_cg	适用于 CG 的溶解效果
dissolve_scene	现将屏幕擦为黑色，然后显示下一场景
dissolve_scene_full	使屏幕自行溶解为黑色，显示另一个场景
dissolve_scene_half	溶解屏幕一段时间，显示下一个场景
wipeleft	从屏幕左侧擦除隐藏当前图像
wipeleft_scene	从屏幕左侧擦除屏幕为黑色，然后显示下一场景
wiperight	从屏幕右侧擦除隐藏当前图像
wiperight_scene	从屏幕右侧擦除屏幕为黑色，然后显示下一场景

zorder 从句 对于 zorder 从句，zorder 后的数字越大，图像在 Z 轴上的距离越远，即离屏幕越远，反之亦然。但由于 DDLC 的内建变换已经为我们设置好了图像的大小与位置，所以 zorder 其实在后续版本不再需要。

临时性变化与对话属性 在视觉小说中，每一次对话角色的神态、姿势可能都会发生变化，那我们岂不是要重复使用很多次 show 语句、hide 语句？所以为了方便开发，Ren'Py 给 say 语句添加了对话属性。

如下例：

```

1 # Chapter 0
2 label ch0_start:
3     show bg club_day
4     "{cps=20}快到{b}学园祭{/b}了。{/cps}{w=.5}{nw}"
5     show monika 1a at t41 zorder 1
6     m "各位！我们得开始准备了！"
7     show sayori 1a at t42 zorder 1
8     s "好耶！！！！"
9     show natsuki 1a at t43 zorder 1
10    n 2d "啊，我都等不及学园祭了。"
11    n "肯定会很棒的！"
12    show yuri 1a at t44 zorder 1
13    y "... "
14    m "那么，是时候来进行分工了。"
15    m 4k "[player]，你想要做什么？"
16    return

```

运行后，我们发现，当夏树说出“啊，我都等不及学园祭了。”会变换一次立绘，以及后面莫妮卡说出 “[player]，你想要做什么？”也会变换立绘，这就是 say 语句的对话属性作用。

不难发现，只需要在角色名后添加立绘编号即可使角色在说出这句话时变换为指定立绘。

临时性变化则是指角色在说出本句前会变换为指定立绘，在本句结束后则换回原来的立绘。要启用临时性变化，只需要在角色名后的立绘编号前加上 @ 即可。如：

```
n @2d "啊，我都等不及学园祭了。"
```

上述例子，在夏树说出“啊，我等不及学园祭了”这句话前与这句话后，都会展示 1a 这个立绘，而当说出这话时，则会变为 2d 这个立绘。

2.2.2.2 hide 语句

hide 语句有着与 show 语句相反的作用——从屏幕上隐藏一个图像。如：

```
hide monika
```

则会将莫妮卡从当前屏幕上隐藏。

hide 语句同样可以使用附加参数，但只能使用 with 从句，使用方法与 show 语句的 with 从句相同，详细请见第 2.2.2.1 小节。

扩展知识

扩展知识：hide 语句同样可以使用 onlayer 从句，用于隐藏对应图层上的图像。但在游戏中很少会涉及到对图层的操作，故本书不会对 onlayer 从句展开讲解。有兴趣者课前往 https://doc.renpy.cn/zh-CN/displaying_images.html 了解详细内容。

2.2.2.3 scene 语句

scene 语句类似于 show 语句，但与 show 语句有一个很明显的差异——使用 scene 语句会清空当前屏幕上的所有图像。scene 语句的显示方式和特性的使用效果与 show 语句一致。

尝试在游戏中运行以下代码，并猜测运行结果：

```

1 # Chapter 0
2 label ch0_start:
3     scene bg club_day
4     with wipeleft_scene
5     "{cps=20}快到{b}学园祭{/b}了。{/cps}{w=.5}{nw}"
6     show monika 1a at 141 zorder 1
7     m "各位！我们得开始准备了！"
8     show sayori 1a at h42 zorder 1
9     s "好耶！！！"
10    show natsuki 1a at t43 zorder 1
11    n 2d "啊，我都等不及学园祭了。"
12    n "肯定会很棒的！"
13    show yuri 1a at s44 zorder 1
14    y "... "
15    scene bg club_day
16    show monika 2a at t11 zorder 1
17    m "那么，是时候来进行分工了。"
18    m 4k "[player]，你想要做什么？"
19    return

```

代码 2.4: script-ch1.rpy

不难发现，执行 `scene` 语句时会将整个屏幕清空。`scene` 语句的语法为：

```
scene <图片编号> <with从句>
```

`with` 从句的使用方法与 `hide`、`show` 语句相同。详细请见[2.2.2.1](#)。

2.2.3 play、stop 语句

现在，在我们的视觉小说中，图像有了，人物立绘与背景也有了，但还是缺了一些东西，比如音乐与音效。在接下来，我们将会进入对于 `play` 和 `stop` 语句的学习。

2.2.3.1 play 语句

`play` 语句主要用于播放音频、音效。请注意，`play` 语句会覆盖当前通道所播放的音频。

Ren'Py 中，我们主要使用三种已经定义好的音频频道：

- `music` - 音乐播放通道；
- `sound` - 声音播放通道；
- `voice` - 语音播放通道。

请阅读以下代码并尝试理解 `play` 语句的语法：

```

1 play music audio.t1
2 play sound audio.closet_open

```

不难看出，`play` 语句的基本语法如下：

```
play <频道名> <文件名>
```

扩展知识

在上述例子中，audio.t1 是一个变量，且被定义为”<loop 22.073>bgm/1.ogg”。故此处的文件名也可以是一个指向文件名的变量。

对于 DDLC 中音乐的定义，您可以查阅 definitions.rpy 中的注释。

播放列表 play 语句的文件名部分也可以是一个列表，且遵循从首到尾的顺序。

读下列代码，尝试理解并运行：

```
1 play music [audio.t1, audio.t2]
2 play sound [audio.closet_open, "<silence .5>", audio.closet_open]
```

特殊效果 play 也可以使用淡出、淡入、循环等效果。

读下列代码，尝试理解并运行。

```
1 play music audio.main_menu loop
2 play music audio.t1 fadein .5 fadeout 1 noloop
```

从实际效果可以看出，loop 可以使一段音频重复播放；fadein 则可以使音频在指定时间内淡入；fadeout 则使音频在指定时间内淡出；noloop 意味不重复播放这段音频。

扩展知识

当 play 语句既没有出现 loop 分句也没有出现 noloop 分句时，Ren'Py 会根据音频的默认配置决定音频的播放。

play 语句同样也可以调整音频的音量。

```
play music audio.t1 volume .55 loop
```

请注意，volume 分句后的值应大于等于 0 小于等于 1

2.2.3.2 stop 语句

stop 语句以关键词 stop 开头，后接想要停止播放的通道名。如：

```
1 stop music fadeout 2
2 stop sound
```

2.2.4 menu 语句

在视觉小说中，往往会有许多选择。这些选择通常能够影响剧情的走向，进入不同的分支。在本节中，你将会学习 menu 语句的基本用法。

读下列代码并尝试运行，看看实际的运行效果：

```
1 # Chapter 0
2 label ch0_start:
3     scene bg club_day
4     "{cps=20}快到{b}学园祭{/b}了。{/cps}{w=.5}{nw}"
5     show monika 1a at 141 zorder 1
6     m "各位！我们得开始准备了！"
7     show sayori 1a at h42 zorder 1
8     s "好耶！！！！"
```

```
9 show natsuki 1a at t43 zorder 1
10 n 2d "啊，我都等不及学园祭了。"
11 n "肯定会很棒的！"
12 show yuri 1a at s44 zorder 1
13 y "... "
14 scene bg club_day
15 show monika 2a at t21 zorder 1
16 show sayori 2a at t22 zorder 1
17 m "那么，是时候来进行分工了。"
18 m 4k "[player]，你想要做什么？"
19 menu:
20     "做小蛋糕":
21         s "夏树的小蛋糕最好吃了！"
22     "布置教室":
23         m "那我们可得抓紧时间了！"
24 return
```

代码 2.5: script-ch1.rpy

menu 的基本语法为：

```
1 menu:
2     "这是一个选项示例" # 文本框内显示的内容。
3     "选项 1": # 选项名称
4         # 选择此项后所的脚本。
5         # 可以添加更多选项。
```



3. Python 与 Ren'Py

本节目标包括：

- 了解 Python 的基本数据类型；
- 学会使用函数与类；
- 学会在 Ren'Py 代码中嵌入 Python 代码；
- 学会使用变量；
- 学会进行判断控制；
- 学会使用等效语句；

Ren'Py 与 Python 是一个不可分割的整体，在 Ren'Py 中，许多复杂的操作都需要依靠 Python 来完成。本章我们将会初步学习如何在 Ren'Py 中使用 Python 语句。

必须注意

本章只会浅显的介绍 Python 的用法，若要真正的学习 Python，您可以前往 <https://www.runoob.com/python3/python3-tutorial.html> 进行更深入的学习。切记，本书不以 Python 为主。

警告

如果您只是想要简单开发一个 Mod，不需要做什么复杂的处理，如只是扩展一下原作剧情，或是给玩家讲述另一个故事，那么您大概率可以跳过本章关于函数、类的学习。但如果您需要开发更复杂的 Mod，比如与 DDLC 有关的 AVG 游戏，或是像 Monika After Story 一样的 Mod，那么函数与类无疑会方便您后期的开发。

但是在学习前，您需要注意一些问题。学习 Python 的难度也比 Ren'Py 要大许多。而且在 Ren'Py 中使用 Python 代码还要格外小心。如在 Python 中，有几种数据类型需要格外注意。这些数据类型轻则导致代码出现意料之外的运行结果，重则使 Ren'Py 不稳定崩溃。

最后，虽然很多 Ren'Py 语句都有等效的 Python 代码，但对于 Mod 来说，不应该使用

Python 代替 Ren'Py 代码。Python 的作用是方便开发者进行更复杂的操作而不用修改 Ren'Py 的底层逻辑。但对于大部分的 Mod 来说，完全可以使用纯 Ren'Py 代码。更不用说 Python 代码在一定程度上会增加脚本的复杂度。Python 学好了，用好了，就是锦上添花；如果用不好，就会造成开发难度直线上升，Debug 及其困难，还会使游戏崩溃给玩家带来负面体验。

3.1 Python 中的几种数据类型

3.1.1 数字

在大多数编程语言中，数字可以分为两类——整型（int）与浮点类（float）型。整型顾名思义，就是指 3、2、6、-2 等不包含小数的数字，浮点型则与之相反，即包含小数的数字，如-2.9、7.1、3.213 等。

数字可以进行运算，如加减乘除。在 Python 中，加法运算使用 + 号，减法运算使用 - 号，乘法运算使用 * 号，除法运算使用 / 号。对于其他的运算符，详细请见表 3.1。如下例：

```

1 >>> 1 + 2
2 # 输出: 3
3
4 >>> 2 / 2
5 # 输出: 1.0
6
7 >>> 10 - 8
8 # 输出: 2
9
10 >>> 2 * 7
11 # 输出: 14

```

扩展知识

我们可以使用变量将想要将数据保存。使用 = 操作符对变量进行赋值，如：

```

>>> a = 1
>>> a
# 输出: 1

>>> a = a * 2
>>> a
# 输出: 2

```

对于乘方、整除、取模（余数）计算，则分别使用 ** 运算符、//运算符与 % 运算符。如下例：

```

1 >>> 1 ** 2
2 # 输出: 2
3
4 >>> 2 ** 3

```

```

5 # 输出: 8
6
7 >>> 10 // 8
8 # 输出: 1
9
10 >>> 14 // 7
11 # 输出: 2
12
13 >>> 14 % 7
14 # 输出: 0
15
16 >>> 25 % 4
17 # 输出: 1

```

3.1.2 字符串

除了数字，Python 还可以处理字符串（str）。在 Python 中可以使用双引号或单引号括起来表示字符串，也可以使用反斜线操作符对特殊字符转义。

```

1 >>> 'Hello'
2 # 输出: 'Hello'
3
4 >>> "Hi!"
5 # 输出: "Hi!"
6
7 >>> I\'m fine.'
8 # 输出: "I'm fine."
9
10 >>> "I'm fine."
11 # 输出: "I'm fine."
12
13 >>> print("This is a backslash: \\")
14 # 输出: This is a backslash: \

```

在 Python 中，字符串同样支持一些运算功能。+ 号能将两个字符串连接起来。* 则用来重复字符串。如：

```

1 >>> 'Hello! ' * 3
2 # 输出: 'Hello! Hello! Hello!'
3
4 >>> "Hi! " + "How are you?"
5 # 输出: "Hi! How are you?"

```

表 3.1: 常用逻辑运算符号与含义

操作符	描述	示例
+	加法运算，将运算符两侧值相加	1 + 2; "str" + "string"

操作符	描述	示例
-	减法运算, 将运算符两侧值相减	1 - 2
*	乘法运算, 将操作符两侧值相称	4 * 2; "str" * 3
/	除法运算, 左操作数除以右操作数	3 / 1
%	取模运算, 左操作数除以右操作数的余数部分	7 % 2
**	幂运算, 左操作数的右操作数次幂	2 ** 3
//	整除运算, 左操作数整除以右操作数	3 // 2
+=	将左侧值加上右侧值并将结果保存给左侧对象	a += 100
-=	将左侧值减去右侧值并将结果保存给左侧对象	a -= 100
*=	将左侧值乘上右侧值并将结果保存给左侧对象	a *= 100
/=	将左侧值除以右侧值并将结果保存给左侧对象	a /= 100
%=	将左侧值取模右侧值并将结果保存给左侧对象	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a 等效于 c = c ** a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

3.1.3 列表与元组

3.1.3.1 列表

在 Python 中, 列表 (list) 是一种常见的数据类型。在列表可以将多种数据组合在一起。如:

```

1 >>> ['1', '2', '3']
2 # 输出: ['1', '2', '3']
3
4 >>> ['This', 'is', 'a', 'list']
5 # 输出: ['This', 'is', 'a', 'list']

```

在列表中, 我们可以使用索引 (index) 来访问指定数据。如:

```

1 >>> name_list = ['Sayori', '莫妮卡', 'Yuri', 'Natsuki']
2 >>> name_list[0]
3 # 输出: 'Sayori'
4
5 >>> name_list[3]
6 # 输出: 'Natsuki'

```

警告

从上例中, 我们可以知道索引从 0 开始。但请注意: 索引最大值不可以超出列表的长度。即在 name_list 这个列表中, 索引最大只能为 3, 因为此时从 0 往后数 4 个数为 3。如果索引最大值超出了列表长度, Python 就会抛出 IndexError 错误。如:

```

>>> name_list = ['Sayori', 'Monika']
>>> name_list[2]

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

```

```
IndexError: list index out of range
```

扩展知识

索引的值也可以为负数，此时 Python 会从列表尾部向前寻找索引。但请注意，若要使 Python 从尾部开始寻找，则索引从-1 开始，且同样不可超出列表最大长度。如下例：

```
>>> name_list = ['Sayori', 'Monika', 'Natsuki', 'Yuri']
>>> name_list[-1]
# 输出: 'Yuri'

>>> name_list[-4]
# 输出: 'Sayori'

>>> name_list[-10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

扩展知识

索引也可用于字符串，但无法修改字符串的数据。字符串中的索引用法与列表中的索引用法相同。

同时，我们可对列表中的数据进行修改。要修改数据，只需要使用索引指定修改的数据，然后使用 = 重新赋值。如：

```
1 >>> name_list = ['Sayori', '莫妮卡', 'Natsuki', 'Yuri']
2 >>> name_list[1] = 'Monika'
3 >>> name_list
4
5 # 输出: ['Sayori', 'Monika', 'Natsuki', 'Yuri']
```

若要添加数据，则可以使用 insert 函数与 append 函数。如

```
1 >>> name_list = ['Monika', 'Sayori']
2 >>> name_list.insert(1, 'Natsuki')
3 >>> name_list.append('Yuri')
4 >>> name_list
5
6 # 输出: ['Monika', 'Natsuki', 'Sayori', 'Yuri']
```

对于 insert 函数，它接受两个参数。第一个参数为一个整数，代表在列表的指定索引处添加一个数据。第二个参数则是添加的数据。

对于 append 函数，它接受一个参数，即要添加的内容。append 函数会在列表末尾添加数据。

若要删除数据，则可以使用 pop 函数与 remove 函数。pop 函数与 remove 函数都只接受一个参数。pop 函数接受一个整数参数，可以删除列表中指定索引处的数据。remove 函数

接受一个任意类型的数据，它会先检查列表中是否存在一个数据与参数相同，如果存在则移除，如果不存在则抛出 `ValueError` 错误。如：

```

1 >>> name_list = ['Monika', 'Sayori', 'Yuri', 'Natsuki']
2 >>> name_list.pop(0)
3 >>> name_list
4 # 输出: ['Sayori', 'Yuri', 'Natsuki']
5
6 >>> name_list.remove("Sayori")
7 # 输出: ['Yuri', 'Natsuki']
8
9 >>> name_list.remove("Monika")
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 ValueError: list.remove(x): x not in list

```

3.1.3.2 元组

元组 (tuple) 是一种不可变的数据类型。元组支持列表除修改、添加、删除外的所有功能。如：

```

1 >>> name_list = ('Sayori', 'Monika', 'Natsuki', 'Yuri')
2 >>> name_list[3]
3 # 输出: 'Yuri'
4
5 >>> name_list.append("Main Character")
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 AttributeError: 'tuple' object has no attribute 'append'

```

3.1.4 字典

字典 (dict) 就像它的名字一样，可以像查字典一样取查找。如：

```

1 >>> point = {'Sayori': 0}
2 >>> point
3 # 输出: {'Sayori': 0}
4
5 >>> point['Sayori'] = 2
6 >>> point['Monika'] = 1
7 >>> point['Natsuki'] = 3
8 >>> point['Yuri'] = 1
9
10 >>> point
11 # 输出: {'Sayori': 2, 'Monika': 1, 'Natsuki': 3, 'Yuri': 1}
12
13 >>> point['Sayori']
14 # 输出: 2

```

3.1.5 布尔类型

布尔类型是最简单的一种类型。布尔类型只包括两个值真 (True)、假 (False)。如：

```

1 >>> is_act_two = False
2 >>> is_act_two
3 # 输出: False
4
5 >>> is_act_two = True
6 >>> is_act_two
7 # 输出: True

```

Python 中的内置数据类型均可进行逻辑运算和比较。如：

```

1 >>> a = False
2 >>> a is True
3 # 输出: False
4
5 >>> not a
6 # 输出: True
7
8 >>> b = 3
9 >>> b != 2
10 # 输出: True
11
12 >>> c = 2
13 >>> c <= b
14 # 即 2 ≤ 3
15
16 # 输出: True

```

详细逻辑操作符请见表3.2。

表 3.2: 常见逻辑、比较运算符与示意

操作符	描述	示例
==	比较两个对象是否相等	2 == 2; a == b
!=	比较两个对象是否不等	1 != 2; a != b
>	比较左对象是否大于右对象	3 > 2
<	比较左对象是否小于右对象	2 < 3
>=	比较左对象是否大于等于右对象	3 >= 2; 3 >= 3
<=	比较左对象是否小于等于右对象	2 <= 3; 3 <= 3
is	比较两个对象内存是否相等 (更加严格的 ==)	a is True
not	逻辑非, 用于反转操作数的逻辑状态。即 True 则为 False, False 则为 True	not True
and	逻辑与, 当只有左操作数与右操作数皆为真时, 条件为真	1 == 2 and 3 < 4

操作符	描述	示例
or	逻辑或, 当左操作数和右操作数中有一个为真时, 条件为真	<code>1 == 2 or 10 > 6</code>

扩展知识

此处只对 Python 的一些常见类型做了简单的介绍。Python 中还有其他类型, 如可调用类 (callable) 等。本书目的以教学 Ren'Py 为主, 故不会涉及 Python 过多。有兴趣者可以前往 <https://www.runoob.com/python3/python3-tutorial.html> 进行更深入的学习。

3.1.6 小节

现在, 让我们将上述所说的 Python 的内置数据类型运用到实际代码中。如下例:

```

1 # Chapter 0
2 label ch0_start:
3     python:
4         # aff - affection
5         n_aff = 0
6         m_aff = 0
7     scene bg club_day
8     "{cps=20}快到{b}学园祭{/b}了。{/cps}{w=.5}{nw}"
9     show monika 1a at 141 zorder 1
10    m "各位! 我们得开始准备了!"
11    show sayori 1a at h42 zorder 1
12    s "好耶! ! ! ! "
13    show natsuki 1a at t43 zorder 1
14    n 2d "啊, 我都等不及学园祭了。"
15    n "肯定会很棒的! "
16    show yuri 1a at s44 zorder 1
17    y "... "
18    scene bg club_day
19    show monika 2a at t21 zorder 1
20    show sayori 2a at t22 zorder 1
21    m "那么, 是时候来进行分工了。"
22    m 4k "[player],你想要做什么? "
23    menu:
24        "做小蛋糕":
25            s "夏树的小蛋糕最好吃了! "
26            python:
27                n_aff += 1
28                m_aff -= 1
29        "布置教室":
30            m "那我们可得抓紧时间了! "
31            python:
32                n_aff -= 1
33                m_aff += 1

```

```
34     return
```

代码 3.1: script-ch1.rpy

上述代码中，我们现将 `n_aff` 与 `m_aff` 赋值为了 0，并且随着玩家做出的选择，`n_aff` 与 `m_aff` 会分别增加或减少一定的数值。

扩展知识

`$`、`python` 语句块都是在 Ren'Py 中运行 Python 代码的方式，将会在后续的章节中介绍。

3.2 函数与类 [Python Only]

3.2.1 函数

在编程中，我们往往会重复执行一段代码或进行类似的操作。为了减少代码的重复，我们可以使用函数。函数的作用就是把相对独立的某个功能抽象出来，成为一个独立的个体。

3.2.1.1 函数的定义

定义一个函数，只需要开头为 `def` 即可。如下例：

```
1 def test(arg1, arg2):
2     print("Arg1 is: " + arg1)
3     print("Arg2 is: " + arg2)
4     return
```

其中，`test` 为这个函数的名字，`arg1`、`arg2` 则为这个函数接受的参数。若留空，代表该函数不接受参数。引号后的部分被称为函数主体，是调用该函数后具体的一些代码。`return` 语句则是函数运行成功后返回的值，可以留空。

调用函数也非常简单，如下例：

```
1 test(1, 2)
2 # 输出: Arg1 is: 1
3 # 输出: Arg2 is: 2
4
5 test(5, arg2=1)
6 # 输出: Arg1 is: 5
7 # 输出: Arg2 is: 1
8
9 test(arg2=4, arg1=2)
10 # 输出: Arg1 is: 2
11 # 输出: Arg2 is: 4
12
13 test(arg1=7, arg2=-7)
14 # 输出: Arg1 is: 7
15 # 输出: Arg2 is: -7
```

由此可见，在给函数传递参数时，可以按照参数的位置直接传递，也可以使用“参数名 = 参数”的方式传递。

3.2.1.2 函数命名空间

命名空间，可以理解成为代码运行的一个环境。函数拥有一个独立于其他代码的运行环境，所有在函数中的变量，都位于一个独立的命名空间内。在这个空间里，函数内部的变量与函数外界的变量不相等。在函数内部调用变量时，会优先去寻找函数的命名空间，再去寻找外部的环境查找变量。因此，函数外的代码无法读取或修改函数命名空间内的变量，函数内也无法直接修改全局命名空间的变量。

如下例：

```

1 >>> a = 1
2 >>> def test():
3 ...     a = 2
4 ...     print(a)
5 >>> print(a)
6 # 输出: 1
7 >>> test()
8 # 输出: 2
9 >>> print(a)
10 # 输出: 1

```

运行上述代码，我们发现虽然一开始我们将 a 这个变量赋值为 1，在函数中，我们将 a “修改”（实际上是在函数内部重新定义了 a）为了 2，并且在函数内部可以看到 a 的值确实为 2，但当我们在函数外打印函数值，会发现值依然是 1，这便是函数命名空间发挥的作用。

如果要在函数内修改外部变量，则应使用 global 语句声明要使用的变量，如下例：

```

1 >>> a = 1
2 >>> def test():
3 ...     global a
4 ...     a = 2
5 ...     print(a)
6
7
8 >>> print(a)
9 # 输出: 1
10 >>> test()
11 # 输出: 2
12 >>> print(a)
13 # 输出: 2

```

运行上述代码，与第一个例子不同在于，在函数内部我们是真真正正的对外部变量 a 进行了修改，并且可以看到最后值是 2，说明修改成功了。

在 Python 中，还有一种情况是函数内部还有一个函数，我们称之为嵌套函数。如下例子：

```

1 def func1():
2     x = 1
3     print(x)
4     def func2():
5         print(x)
6
7     def func3():

```

```

8     x = 2
9     print(x)
10    func2()
11    func3()
12    print(x)

```

在这个例子中，func2、func3 就被称为嵌套函数。嵌套函数符合函数的特性，拥有一个独立的、不与其他代码共享的运行空间。但是，嵌套函数也能访问定义这个嵌套函数范围内的变量而不需声明。因此，这段代码的运行结果如下：

```

1 >>> func1()
2 # 输出: 1
3 # 输出: 1
4 # 输出: 2
5 # 输出: 1

```

但请注意，嵌套函数有一种特殊情况无法访问范围外变量，当嵌套函数内的代码存在一个同名变量或者在后面的代码中对这个变量进行了修改，那么 Python 会认为一开始的这个代码企图在变量定义之前访问，会抛出 `UnboundLocalError` 错误。具体例子如下：

```

1 def func1():
2     x = 0
3     def func2():
4         print(x)
5         x = 1
6         print(x)
7     print(x)

```

我们预期中的运行结果应该输出 0 1 0。但事实上，这段代码会抛出 `UnboundLocalError` 错误。原因就是在 `func2` 中，我们定义了一个与 `func1` 中的 `x` 变量同名的变量，当 Python 运行到第 4 行时，由于第 5 行对变量 `x` 进行了定义，它会认为我们想要访问的应该是 `func2` 中的变量 `x` 而不是 `func1` 中的变量 `x`，而 `func2` 中的变量 `x` 是在第 5 行才被定义，故 Python 会抛出错误，告诉我们我们企图在变量 `x` 定义之前访问 `x` 这个变量。

这个时候如果我们想在嵌套函数内修改外部环境中的变量的值该怎么办呢？答案是 `nonlocal` 语句，它和 `global` 语句的用法一致，但是语义不同。如下例：

```

1 x = 0
2 y = 3
3 print(x)
4 def func1():
5     global x
6     x = 1
7     print(x)
8     y = 0
9     def func2():
10        nonlocal y
11        y = 2
12        print(y)
13    func2()
14    print(y)

```

```

15
16 func1()
17 print(x)
18 print(y)

```

这段代码的运行结果如下：

```

1 # 输出: 0
2 # 输出: 1
3 # 输出: 2
4 # 输出: 2
5 # 输出: 1
6 # 输出: 3

```

上述代码中，我们在全局范围内定义了 `x`、`y` 两个变量并分别赋值为 0、3。在 `func1` 中，我们使用 `global` 语句声明了 `func1` 函数内部的 `x` 变量就指向外部变量 `x`，并修改为了 1，并在 `func1` 中定义了内部变量 `y`。随后，我们运行到了 `func2` 函数中。在 `func2` 函数中，我们通过 `nonlocal` 语句声明了 `func2` 中的变量 `y` 指向的是内部变量 `y`（注意不是外部变量 `y`）并修改值为 2。在 `func1` 中打印变量 `y`，也会发现值为 2，说明 `func2` 函数中的 `y` 确实指向了 `func1` 函数中的 `y`。最后在外部打印 `x`、`y` 变量，发现 `x` 被修改为 1，而 `y` 依然为 3，说明 `func1` 中 `x` 确实是外部变量 `x`，`func1`、`func2` 中的变量 `y` 与外部变量 `y` 不相同。

3.2.2 类

Python 是一门面向对象的语言，而类是一种用来描述具有相同属性和方法（函数）的对象的集合。它定义了该集合中每个对象共同具有的方法。对象是类的实例化，实例化就是用类创建对象的过程。简单来说，类就是一个模板，存放着一堆特殊的函数，而实例化就是用这个模板照葫芦画瓢生成一个对象的过程。对象具有唯一性，每一个实例化结果的对象都是不同的。

类中的函数分为三种：类函数、方法函数、静态函数。静态函数与普通函数并无差异，而类函数与方法函数属于绑定函数，它们和类与对象息息相关。在目前阶段，我们并不需要去了解每种函数的定义以及作用，我们只需要明白类中存放着这三种函数即可。

3.2.2.1 何时使用类？

专业术语来说，类是面向对象的，而函数是面向过程的。

类将多个功能相关的函数进行封装，是多个函数的集合，可以生成多个独一无二的对象，每个对象之间互不相同、对象中的变量也不相同。类可以保存某个属性的状态，同时也可理解成为一个模板，照着这个模板，我们能生成一个对象。并且类具有修改容易的优点。

那么何时使用类呢？如下例：

```

1 # 定义一个字典，存放每个角色支线故事是否解锁的状态
2 unlocked = {'monika': False, 'natsuki': False, 'yuri': False, 'sayori': False}
3
4 # 定义跳转到对应角色支线故事的函数
5 def jump_to_monika():
6     ...
7 def jump_to_natsuki():
8     ...

```

```

9 def jump_to_sayori():
10     ...
11 def jump_to_yuri():
12     ...
13
14 # 定义开始支线故事的函数
15 def side_story(winner, winner_point):
16     # 声明 unlocked 变量是一个全局变量
17     global unlocked
18
19     # target_points: 目标分数, 如果分数没有达到目标分数则不开启支线故事
20     target_points = 10
21
22     # 判断
23     if winner_point >= target_point:
24         if winner == "monika":
25             jump_to_monika()
26         elif winner == 'natsuki':
27             jump_to_natsuki()
28         elif winner == 'sayori':
29             jump_to_sayori()
30         elif winner == 'yuri':
31             jump_to_yuri()
32
33     # 设置状态
34     unlocked[winner] = True
35
36 side_story('monika', 5)

```

可以看到, 如果用函数来实现, 我们需要在外部环境定义许多函数与变量, 例如 `unlocked`、`jump_to_monika` 等。稍有不当, 我们就会污染整个外部环境。并且因为我们无法直接在游戏中对函数内的变量进行修改, 所以我们没有办法在游戏运行过程中降低目标分数。如果用类来实现:

```

1 # 定义一个类
2 class SideStory:
3
4     # __init__ 是一个特殊的函数, 它用于实例化时告诉 Python 要怎样生成对象
5     def __init__(self):
6         # 设置 unlocked 变量
7         self.unlocked = {'monika': False, 'natsuki': False, 'yuri': False,
8                         'sayori': False}
9
10        # target_points: 目标分数, 如果分数没有达到目标分数则不开启支线故事
11        self.target_points = 10
12
13    def start(self, winner, winner_point):
14        # 判断
15        if winner_point >= self.target_points:

```

```

15     if winner == "monika":
16         ...
17     elif winner == 'natsuki':
18         ...
19     elif winner == 'sayori':
20         ...
21     elif winner == 'yuri':
22         ...
23
24         self.unlocked[winner] = True
25
26 side_story = SideStory()
27 side_story.start('monika', 5)
28
29 side_story.target_points = 5

```

我们可以看到，使用类与对象，我们就避免了在外部环境定义许多函数与变量，它们都被放在了类的命名空间中，避免了污染外部命名空间。并且类还有一个好处：容易在外部对内部变量进行修改，因此对于一开始的问题便得到了解决。

扩展知识

`self` 是一个特殊的参数。当类被实例化后，无论调用其中的哪一个方法（静态方法除外），Python 都会将这个对象自己传递给第一个参数。

3.2.2.2 类的定义

定义一个类，只需要开头为 `class` 即可。如下例：

```

1 class Test:
2     def __init__(self):
3         self.a = 1
4
5     def counter(self):
6         self.a += 1
7         print(self.a)

```

上述例子中，定义了一个名为 `Test` 的类，这个类中有一个变量为 `a`，且有一个 `counter` 方法用于打印 `a` 的值。

扩展知识

在类中，以双下划线开头的、具有特殊的方法名的方法叫魔法方法（Magic Methods）。上述例子中的 `__init__` 特殊方法用于在实例化一个类时会运行的初始化函数。类似的魔法方法还有 `__eq__`, `__ne__` 等。

3.2.2.3 类的使用

一般来说，在使用类前，需要对类进行实例化。实例化后，类就会变成对象。创造对象和创造变量类似。如下例：

```

1 >>> test = Test()
2 >>> test.counter()
3 # 输出: 2
4 >>> test.counter()
5 # 输出: 3

```

第一行的“test = Test()”创造了一个Test对象。剩下两行代码则是在调用这个对象的counter方法。

如果要在类的方法中定义或使用一个属性，必须使用“self.<变量名> = <值>”的方式进行赋值，否则这个属性就只会存在于方法的命名空间而不是对象的命名空间。简单理解就是如果不使用上述方式赋值，那么这个值就不会被保存到对象里面。

扩展知识

Python 中类与函数的使用远远不止这些，您可以前往<https://www.runoob.com/python3/python3-function.html> 与 <https://www.runoob.com/python3/python3-class.html> 了解更多。

3.3 在 Ren'Py 中使用 Python 语句

在 Ren'Py 中有多种方式可以使用 Python 语句：Python 语句块（block）、单行 Python 语句、init python 语句。

3.3.1 Python 语句块

Python 语句块是使用 Python 的最方便的一种形式。一个 Python 语句块包含两个部分：

- 开头的声明
- Python 代码

如下例：

```

1 # Chapter 0
2 label ch0_start:
3     python:
4         # affection: 好感度
5         n_aff = 0
6         m_aff = 0
7         scene bg club_day
8         "{cps=20}快到{b}学园祭{/b}了。{/cps}{w=.5}{nw}"
9         show monika 1a at 141 zorder 1
10        m "各位！ 我们得开始准备了！"
11        show sayori 1a at h42 zorder 1
12        s "好耶！！！！"
13        show natsuki 1a at t43 zorder 1
14        n 2d "啊， 我都等不及学园祭了。"
15        n "肯定会很棒的！"
16        show yuri 1a at s44 zorder 1
17        y "... "
18        scene bg club_day

```

```

19     show monika 2a at t21 zorder 1
20     show sayori 2a at t22 zorder 1
21     m "那么，是时候来进行分工了。"
22     m 4k "[player]，你想要做什么？"
23     menu:
24         "做小蛋糕":
25             s "夏树的小蛋糕最好吃了！"
26             python:
27                 $ n_aff += 1
28                 $ m_aff -= 1
29         "布置教室":
30             m "那我们可得抓紧时间了！"
31             python:
32                 $ m_aff += 1
33                 $ n_aff -= 1
34     return

```

当出现多个 Python 语句块时，Ren'Py 会根据先后顺序依次执行。同时，我们可以使用 `hide` 与 `in` 关键词来改变 Python 语句块的行为。

`hide` 关键词会使 Python 语句块在一个独立的环境（类似于函数命名空间）下运行，即不与其他 Python 语句块共用一个环境。在具有 `hide` 关键词的 Python 语句块中的所有变量将不会被保存。

`in` 关键词则可以让 Python 语句块在一个独立的环境下运行。其他环境可以通过“储存区. 变量”来使用其他环境的内容。

如下例：

```

1 python:
2     a = 2
3
4 python hide:
5     a = 1
6
7 python in c:
8     a = 4
9
10 python:
11     print(c.a)
12     print(a)

```

最后的运行结果为：4、2。

3.3.2 单行 Python 语句

大多数情况下，我们只有一行 Python 语句需要执行。此时使用 Python 语句块无疑会对开发者造成多余的输入。为了让编写只有一行的 Python 更方便快捷，Ren'Py 提供了单行 Python 语句。

单行 Python 语句以美元符号 (\$) 开头。如：

```

1 $ s_aff = 0
2

```

```
3 $ winner = 'monika'
```

3.3.3 init python 语句

init python 语句在 Ren'Py 初始化阶段运行，会早于其他代码。这种功能可以用于定义类和函数或者配置变量。

扩展知识

在 **init** 与 **python** 之间还可以放一个运行优先级，默认为 **0**。**init** 语句将会按照从低到高的顺序执行。即优先级为**-1** 的代码会优先于优先级为 **0** 的代码执行。在优先级相同的情况下，Ren'Py 会根据本代码所在文件的 **Unicode** 码顺序执行。即在代码优先级都在 **0** 的情况下，**a.rpy** 内的代码总会优先于 **b.rpy** 内的代码执行。

必须注意

为了避免与 Ren'Py 自身代码冲突，您只应使用**-999** 到 **999** 范围内作为优先级。同时，原则上除特殊代码外，**init** 优先级应全部大于等于 **0**。

init python 语句也可以使用 **hide** 或 **in** 分句，与普通 python 语句用法相同。

在 init python 语句中的变量不会被存档。因此，在 init 语句中定义的应为常量。

```
1 # Chapter 0
2 init -1 python:
3     demo = True
4
5 label ch0_start:
6     python:
7         # affection: 好感度
8         n_aff = 0
9         m_aff = 0
10    scene bg club_day
11    "{cps=20}快到{b}学园祭{/b}了。{/cps}{w=.5}{nw}"
12    show monika 1a at 141 zorder 1
13    m "各位！ 我们得开始准备了！"
14    show sayori 1a at h42 zorder 1
15    s "好耶！！！！"
16    show natsuki 1a at t43 zorder 1
17    n 2d "啊，我都等不及学园祭了。"
18    n "肯定会很棒的！"
19    show yuri 1a at s44 zorder 1
20    y "... "
21    scene bg club_day
22    show monika 2a at t21 zorder 1
23    show sayori 2a at t22 zorder 1
24    m "那么，是时候来进行分工了。"
25    m 4k "[player]，你想要做什么？"
26
27 if demo:
```

```

28     "Demo 版剧情到此结束。"
29     return
30
31     menu:
32         "做小蛋糕":
33             s "夏树的小蛋糕最好吃了！"
34             python:
35                 $ n_aff += 1
36                 $ m_aff -= 1
37         "布置教室":
38             m "那我们可得抓紧时间了！"
39             python:
40                 $ m_aff += 1
41                 $ n_aff -= 1
42
43     return

```

扩展知识

Python 语句块还可以使用 `early` 分句。在 Ren'Py 的运行生命周期中，`python early` 是被最早运行的。`python early` 中的代码将会早于所有代码运行，因此，`python early` 语句适合用来修改 Ren'Py 底层处理机制。但由于修改底层处理代码可能会导致一系列问题，我们不会在这里展开讲解。有兴趣者可以前往 https://doc.renpy.cn/zh-CN/1_ifecycle.html 了解 `python early` 语句的使用。

3.4 使用变量

3.4.1 python 语句块

关于如何在 Python 语句块中定义变量，请阅读第3.3章

3.4.2 define 语句

`define` 语句在初始化时将一个变量赋值。此变量视为一个常量，初始化之后不应再改变。例如：

```

1 define demo = True
2 define isActTwo = False

```

这段代码的运行效果等于：

```

1 init python:
2     demo = True
3     isActTwo = False

```

扩展知识

在 `define` 语句中同样可以指定优先级，只需要在 `define` 关键词后添加优先级即可，如：`define 3 demo = True`。

define 还可以为我们创建一个储存区，只需要将 define 关键词后的变量改为“储存区. 变量”的形式即可。

3.4.3 default 语句

default 语句会给一个未被定义的变量初始赋值。default 语句适合用来定义在游戏过程中会变化的变量。如下例：

```
default demo = False
```

当 demo 这个变量在游戏开始后没有被定义，则将等价于在 start 脚本标签中定义 demo，且值为 False。若在存档加载后没有被定义，则等价于在 after_load 魔法标签中定义 demo，且值为 False。总而言之，若 demo 这个变量在游戏开始时没有被定义，那么它的值就是 False，除非在后来的代码中它的值被改变了。

3.4.4 持久化数据

持久化数据是 Ren'Py 中的一个储存区，无论用户怎样存档、读档，持久化数据区的数据总是独立于 Ren'Py 的游戏存档数据的。如 DDLC 中，对于用户是否在一周期走过了每一条支线、解锁了每一个 CG 的字典（dict），就存储在持久化数据中，避免因读档存档导致数据消失。简单来说，持久化数据就是不会随着用户存档、读档而改变的数据。

一般来说，在使用 DDLC 中文 Mod 模板制作且没有对模版进行设置的游戏，在 Ren'Py 标准位置里会有一个名为 DDLCModTemplateZh 的文件夹，里面通常有一个名为 persistent 的文件，那就是存储持久化数据的存档文件。

持久化数据的用法就和普通变量一样，只不过在前面需要加上“persistent.” 前缀。简单来说，持久化数据的语法如下：

```
1 persistent.<变量名> = <Python 数据类型>
```

例如：

```
1 default persistent.monika_deleted = False
```

无论用户如何重启游戏、读档、存档，除非在代码中对持久化数据进行修改或删除 persistent 文件，persistent.monika_deleted 的值永远都只会是 False。

3.5 流程控制

流程控制控制了程序运行的步骤。流程控制包括顺序控制、条件控制和循环控制。顺序控制，顾名思义，就是按照代码的先后顺序，从上到下依次执行代码。

3.5.1 脚本标签

在 Ren'Py 中，我们可以使用 label 语句，用自定义的标签名声明一个程序点位。这些标签用于调用或者跳转，可以使用在 Ren'Py 脚本、Python 函数及各类界面中。

3.5.1.1 label 语句

在游戏中，故事常常会有多个走向，这是我们就要编写多个分支。如果剧情只围绕一个分支来讲述故事，那么一定是很枯燥的。同时，如果我们把所有的代码都写在一个 label 里，无疑会对编写与后期的维护造成不必要的麻烦。这时候，就需要定义多个 label。

label 语句的基本语法为：

```

1  label <标签名>(参数 1, 参数 2, ...):
2      <语句 1>
3      <语句 2>
4      <语句 3>
5      ...

```

当标签不接受参数时，可以省略小括号及里面的内容。如下例子：

```

1  label ch0_end:
2      scene bg club_day
3      "多么美好的一天啊！"
4      return

```

警告

通常在 label 语句末尾，我们都会使用 return 来返回到上一个调用栈（stack）。在这里您可以理解为回到之前执行的函数、label 继续运行游戏。

如果没有 return 语句，在执行完本 label，Ren'Py 会继续调用在本 label 定义之后的 label。

label 可以在不同的文件内定义。例如我们现在在 game 目录下创造一个名为 script-ch0_tasks.rpy 的文件，并向这个文件中写入以下内容：

```

1  label ch0_monika:
2      scene bg club_day
3      show monika 1a at t11
4      m "想好要做什么了吗？"
5      return
6
7  label ch0_natsuki:
8      scene bg club_day
9      show natsuki 1a at t11
10     n "不过，你知道怎么做小蛋糕吗？"
11     return

```

代码 3.2: script-ch0_tasks.rpy

此时，我们在 ch0_start 标签中可以调用 ch0_monika 与 ch0_natsuki 标签。

3.5.1.2 call 语句与 jump 语句

现在，让我们修改一下 script-ch0.rpy 中的内容：

```

1  default n_aff = 0
2  default s_aff = 0
3  default demo = False
4
5  label ch0_start:
6      scene bg club_day
7      "{cps=20}快到{b}学园祭{/b}了。{/cps}{w=.5}{nw}"
8      show monika 1a at 141 zorder 1

```

```

9   m "各位！ 我们得开始准备了！ "
10  show sayori 1a at h42 zorder 1
11  s "好耶！ ! ! "
12  show natsuki 1a at t43 zorder 1
13  n 2d "啊， 我都等不及学园祭了。"
14  n "肯定会很棒的！ "
15  show yuri 1a at s44 zorder 1
16  y "... "
17  scene bg club_day
18  show monika 2a at t21 zorder 1
19  show sayori 2a at t22 zorder 1
20  m "那么， 是时候来进行分工了。"
21  m 4k "[player]， 你想要做什么？ "
22
23  if demo:
24      "Demo 版剧情到此结束。"
25      return
26
27  menu:
28      "做小蛋糕":
29          s "夏树的小蛋糕最好吃了！ "
30          python:
31              $ n_aff += 1
32              $ m_aff -= 1
33              call ch0_natsuki
34      "布置教室":
35          m "那我们可得抓紧时间了！ "
36          python:
37              $ m_aff += 1
38              $ n_aff -= 1
39              call ch0_monika
40      return

```

代码 3.3: script-ch0.rpy

运行上述代码，我们会发现在玩家做出选择后，执行了在 script-ch1_tasks.rpy 中的内容。这就依赖于 call 语句和 jump 语句为我们提供的跳转功能了。

call 语句和 jump 语句可以将程序跳转到一个指定的脚本标签处。

call 与 jump 的区别在于，call 语句相当于调用，在目标标签执行完毕后会回到当前标签。而 jump 相当于跳转，在目标标签执行完后将不会返回当前标签。

call 语句和 jump 语句的语法如下：

```
1 call/jump <标签名>
```

或者

```
1 call/jump expression <label 表达式>
```

如下例：

```
1 label ch0_start:
2     scene bg club_day
```

```

3   m "你现在正在主标签内。"
4   $ today_winner = "sayori"
5   call test_natsuki
6   m "哦， 你回来了？（跳转到 test_natsuki 标签后返回主标签。）"
7   call expression "test_" + today_winner
8   m "你刚刚又去哪里了？（跳转到 test_sayori 标签后再次返回主标签。）"
9   jump no_way
10  m "[player]， 你还在吗？（这行不会被执行）"
11  return
12
13 label test_natsuki:
14   n "你跳转到了 test_natsuki 标签内。"
15   return
16
17 label test_sayori:
18   s "你跳转到了 test_sayori 标签内。"
19   return
20
21 label no_way:
22   y "好吧， 看起来你回不去了。（因为使用了jump， 所以你无法回到莫妮卡那里了）"
23   return

```

运行上述代码，我们会发现我们一开始会运行 ch0_start 标签中的内容。接着，我们会跳转到 test_natsuki 标签内并运行代码，运行完成后我们会返回到 ch0_start 标签中，随后再次跳转到 test_sayori 标签内，然后我们又回到了 ch0_start 标签中，最后，我们跳转到了 no_way 标签中，并且不再返回 ch0_start 标签，而是回到开始界面。

你或许注意到了，在 ch0_start 中我们并没有直接使用 call test_sayori 语句，而是使用了一个简单的表达式，然后把这个运算结果传递给了 call。这就是 expression 选项的作用。使用 expression 选项，我们不用把标签名写死在程序里，可以立即运算表达式的结果并传递给 call。这样做的好处是可以方便地在同一日的多个分支中跳转。

3.5.2 if 判断

大多数游戏中都具有多条剧情线。但面对一些只希望玩家触发多条剧情中的一条时，我们就可以利用 if 语句判断玩家的剧情线路。在 Python 和 Ren'Py 中，if 语句的基本语法为：

```

1 if <表达式>:
2     <语句 1>
3     <语句 2>
4 elif <表达式>:
5     <语句 3>
6 elif <表达式>:
7     <语句 4>
8 elif ....:
9     ...
10 else:
11     <语句 5>

```

每一个 if 语句中的表达式都应当返回一个 True 或 False（见表3.2）。结果为 True 时，将

会执行 if 语句块中的代码，如果结果为 False，Python 就会忽略 if 语句块内的所有代码。

扩展知识

表达式也可以是一个数字、一个字符串、或定义了 `__bool__` 魔术方法的对象。

不为 0 的数字、非空的字符串以及 `__bool__` 方法返回 `True` 的对象都会被视为 `True`，而 `None`、空字符串、0，当然还有 `False` 本身则会被视为 `False`

如下例：

```

1 if 1:
2     print('1 is True.')
3
4 x = True
5
6 if x:
7     print('x is True.')
8
9 if 1 + 1 == 2:
10    print('Math is still correct.')

```

上述代码的输出结果为：

```

1 1 is True.
2 x is True.
3 Math is still correct.

```

当表达式存在多个可能时，则可以使用 `elif` 语句。一个 `if` 语句中可以存在多个 `elif` 语句。如果想要在所有的 `if`、`elif` 语句都不成立时，额外执行一些代码，则可以使用 `else` 语句。

当 `if` 语句中的表达式为 `False` 时，会先判断 `elif` 子句是否成立，成立则执行该子句下的代码，否则继续判断其他 `elif` 子句，直到运行到 `else` 子句。如果运行到 `else` 子句时，`if` 和 `elif` 语句的表达式中没有成立的均为 `False`，那么就会执行 `else` 语句的内容。请注意，`elif` 或 `else` 必须与 `if` 连用。如下例：

```

1 if 1 + 1 != 2:
2     print('Math crashes!')
3 elif 1 + 2 == 2:
4     print('Math crashes again!')
5 else:
6     print("It's OK. Nothing crazy happened.")

```

3.5.3 循环

循环允许我们重复执行一段代码而不需要编写更多的代码。Python 中存在两种循环：`while` 循环与 `for` 循环。

3.5.3.1 while 循环

`while` 循环是 Python 和 Ren'Py 中最简单的循环。它的语法结构如下：

```

1 while <表达式>:
2     <语句 1>

```

```

3 <语句 2>
4 ...

```

`while` 循环的表达式与 `if` 循环的表达式一样。只有表达式为 `True` 时才会执行 `while` 内的语句。例如：

```

1 i = 0
2 while i < 10:
3     print(i)
4     i += 1

```

输出结果为：

```

1 0
2 1
3 2
4 3
5 4
6 5
7 6
8 7
9 8
10 9

```

警告

请注意，一般表达式不为 `True`，否则就会出现无限循环或死循环。如将上例中的 `i+=1` 删去，就会导致 `i` 这个变量恒定为 `0`，`0` 又一定会小于 `10`，那么该表达式就一定会返回 `True`，使得程序卡死在 `while` 循环，无法运行下一个代码。在后文中我们将会介绍 `break` 和 `continue` 语句来打破或跳过循环。

3.5.3.2 for 循环 [Python Only]

`for` 循环比 `while` 循环的使用方法更加丰富。它的语法结构如下：

```

1 for <变量名> in <序列>:
2     <语句 1>
3     <语句 2>
4 ...

```

这里的序列可以是列表、元组等可迭代对象。当序列中不再有变量后，`for` 循环会停止运行。如下例：

```

1 t1 = ('Sayori', 'Monika', 'Yuri', 'Natsuki')
2 l1 = [0, 0, 2, 0]
3
4 for i in t1:
5     print(i)
6
7 for i in l1:
8     print(i)

```

输出结果为：

```
1 Sayori  
2 Monika  
3 Yuri  
4 Natsuki  
5 0  
6 0  
7 2  
8 0
```

扩展知识

`range` 函数是 Python 的内部函数之一，它可以为我们生成一个生成器（类似于列表，但比列表的性能更好）。使用 `range` 函数，我们可以快速生成一个从某数开始，按照一定规律排列到某数结束的一个生成器。如下例：

```
1 for i in range(10):  
2     print(i)  
3  
4 print("=====")  
5  
6 for i in range(10, 0, -1):  
7     print(i)
```

输出结果为：

```
1 0  
2 1  
3 2  
4 3  
5 4  
6 5  
7 6  
8 7  
9 8  
10 9  
11 =====  
12 10  
13 9  
14 8  
15 7  
16 6  
17 5  
18 4  
19 3  
20 2  
21 1
```

上述第二个例子中的 **10** 就是起始数字，**0** 则为截止数字，**-1** 就是步长。`range` 会从 **10** 开始，依次加上**-1**，即减去 **1**，直到运算结果为 **0** 时停止。（请注意，截止数字不会包含在 `range` 所生成的生成器中）

在部分情况下，我们希望可以跳过循环或退出循环体，这时我们就可以使用 `break` 和 `continue` 语句了。`break` 可以立即退出循环体，如下例：

```

1 i = 0
2 while i < 10:
3     print(i)
4     i += 1
5     if i > 5:
6         break
7 print("End")

```

输出结果为：

```

1 0
2 1
3 2
4 3
5 4
6 5
7 End

```

可以看到，在这个过程中 `i` 的值为 0 到 5，虽然 `i<10` 此时依然成立，但由于 `while` 循环下编写了一个 `if` 判断，当 `i` 大于 5 时执行 `break` 语句，打破了循环，所以没有输出 6、7、8、9。`continue` 可以跳过当前的循环，如下例：

```

1 for i in range(10):
2     if i == 5:
3         continue
4     print(i)

```

输出结果为：

```

1 0
2 1
3 2
4 3
5 4
6 6
7 7
8 8
9 9

```

可以看到输出结果中并没有 5，这是由于 `for` 循环下的 `if` 语句判断当 `i` 等于 5 时，执行 `continue` 子句，`continue` 子句会让后续代码被跳过，直接进入下一个循环，所以没有输出 5。

同时，在 Python 中，循环也可以使用 `else` 语句。在 `while` 语句中的 `else` 语句会在 `while` 表达式为 `False` 时被执行。如下例：

```

1 i = 0
2 while i <= 9:
3     print(i)
4     i += 1
5 else:
6     print(i, " is bigger than 9")

```

输出结果为：

```

1 0
2 1
3 2
4 3
5 4
6 5
7 6
8 7
9 8
10 9
11 10 is bigger than 9

```

同时，请注意 break 导致的循环体退出不会执行 else 语句中的内容。

3.5.4 错误和异常 [Python Only]

在 Python 中，不正常或语法错误的代码将会抛出异常。异常会使程序停止运行、崩溃、闪退等。常见的异常有：NameError（尝试使用一个未定义的变量）、IndexError（尝试访问在列表或元组范围外的索引）、TypeError（试图将两个不支持运算的类型进行运算）等。

为了处理这些异常，我们可以使用 try-except 语句。它的语法结构如下：

```

1 try:
2     <可能抛出错误的语句>
3 except <错误类型>:
4     <当错误被捕获后的语句>

```

如下例子：

```

1 try:
2     x = 1 / 0
3 except ZeroDivisionError:
4     print('Error: Cannot divide by zero.')

```

输出结果为：

```
1 Error: Cannot divide by zero.
```

进阶的语法包含 finally 从句或 else 从句。请注意，finally 从句与 else 从句不可并存。

在 try-except-finally 中，无论 try 代码块中的代码是否抛出异常，finally 从句中的代码都一定会被执行。try-except-finally 的语法如下：

```

1 try:
2     <可能抛出错误的语句>
3 except <错误类型>:
4     <当错误被捕获后的语句>
5 finally:
6     <无论是否抛出错误都会执行的语句>

```

如下例子：

```

1 div = 0
2 def func():

```

```

3     global div
4         return 1 / div
5
6 try:
7     x = func()
8 except ZeroDivisionError:
9     print('Error: Cannot divide by zero.')
10 finally:
11     x = 10
12
13 print(x)
14
15 div = 10
16 try:
17     x = func()
18 except ZeroDivisionError:
19     print('Error: Cannot divide by zero.')
20 finally:
21     x = 100
22 print(x)

```

输出结果为：

```

1 Error: Cannot divide by zero.
2 10
3 100

```

try-except-else 中，只有当 try 语句块中的内容没有抛出异常时，else 中的内容才会被执行。try-except-else 的语法如下：

```

1 try:
2     <可能抛出错误的语句>
3 except <错误类型>:
4     <当错误被捕获后的语句>
5 else:
6     <当没有错误被捕获时执行的语句>

```

如下例：

```

1 try:
2     x = 1 / 0
3 except ZeroDivisionError:
4     print('Error: Cannot divide by zero.')
5 else:
6     print("The value of x is ", x)
7
8 try:
9     x = 1 / 10
10 except ZeroDivisionError:
11     print('Error: Cannot divide by zero.')
12 else:
13     print("The value of x is ", x)

```

输出结果为：

```
1 Error: Cannot divide by zero.  
2 The value of x is 0.1
```

3.6 等效语句

等效语句是在 Python 中使用 Ren'Py 语法的一种方式。等效语句只能在 Python 代码中运行。

3.6.1 对话

Ren'Py 的 say 语句对应两种等效语句。如下例：

```
1 m "你好！"
```

这段代码不仅等效于下列代码：

```
1 $ m("你好！")
```

同时也等效于：

```
1 $ renpy.say(m, "你好")
```

不过，为了保持语义上的通畅，我们常使用后者。

3.6.2 图像显示

3.6.2.1 show

show 语句的等效语句的语法如下：

```
1 renpy.show(<name>, at_list=<position>, zorder=<zorder number>)
```

3.6.2.2 hide

hide 语句的等效语句的语法如下：

```
1 renpy.hide(<name>)
```

3.6.2.3 scene

scene 语句的等效语句的语法如下：

```
1 renpy.scene()  
2 renpy.show(<name>)
```

3.6.2.4 with 从句

with 语句的等效语句的语法如下：

```
1 renpy.with_statement(<name>)
```

3.6.3 call 和 jump

call 和 jump 语句的等效语句的语法如下：

```
1 renpy.call(script=<script name>)
2 renpy.jump(screen=<script name>)
```

对于更多关于等效语句的介绍, 请查阅 https://doc.renpy.cn/zh-CN/statement_equivalents.html



4. 增添资源

本节目标包括：

- 学会增添图像、音频资源；
- 学会使用 `image` 语句定义图像；
- 学会定义音频

在开发中，我们往往会想增添一些资源，如更多的背景音乐、更丰富的角色表情与背景、为角色增添更多服装等。在本章中，我们将重点学习如何管理模组资源，往模组中增添新的图像、音频等。

4.1 增添资源

一般来说，在 Mod 工程里的 `game` 文件夹下会有一个名叫 `mod_assets` 的文件夹。所有在 `mod_assets` 文件夹里的内容都会在生成发行版时被打包成一个 `rpa` 文件。（在后续的章节里，我们将会详细学习如何控制哪些文件打包、哪些不打包，以及如何生成发行版）

扩展知识

目前，常见的资源获取方法有：从 DDLC Community Assets 获取（Google Drive）、从一些开发 QQ 群提供的资源、Reddit 等。

必须注意

在使用资源前，请务必注意版权问题。一般来说，在 DDLC Community Assets 中的文件只需要在感谢名单中添加作者的名字，但也有部分资源会有更多的要求。请记住：如果没有得到作者的授权就使用资源是侵权违法行为。保护作者的知识产权，不仅是在维护作者的权利、保护作者的心血，更是在保护创造、促进 DDLC 社区的良好发展。

4.1.1 定义角色

在一些模组中，我们常常需要添加一些角色来丰富故事线，或是推动故事情节的发展。要增加角色，我们需要先了解如何定义一个角色。

定义一个角色有两种方式：`Character` 与 `DynamicCharacter`。两种定义方式几乎没有区别，唯一的区别在于 `Character` 的名字是固定的，无法更改的；而 `DynamicCharacter` 则使用一个变量作为角色名，是动态的，可以更改。由于 DDLC 中角色都使用 `DynamicCharacter` 来定义角色，故本书不会介绍如何使用 `Character` 定义，感兴趣者可以前往 Ren'Py 中文文档了解。

`DynamicCharacter` 的语法如下：

```
1 define <变量名> = DynamicCharacter('<存储角色名的变量名>', image='<say语句的对话属性图像名>', what_prefix='"', what_suffix='"', ctc="ctc", ctc_position="fixed")
```

警告

对于 `say` 语句的对话属性来说，如果 `image` 后的图像名错误，那么将会导致其无法使用。所以请务必确保 `image` 后的图像名与您定义的角色立绘名一致。对于角色立绘名，请参考[2.2.2.1](#)

扩展知识

在某些情况下，我们可能改变角色说话内容的双引号为其他符号，我们可以通过修改 `what_prefix` 和 `what_suffix` 实现效果。

正常情况下，除旁白外所有的角色说的话都会被双引号包住。如：

```
1 m "你好"
```

那么在游戏中的效果为：“你好”。如果我们想要修改为：「你好」，那么只需要找到莫妮卡角色的定义，并将 `what_prefix` 改为「，`what_suffix` 改为」，即可实现。实际代码如下：

```
1 define m = DynamicCharacter('m_name', image='monika', what_prefix='「',
    what_suffix='」', ctc='ctc', ctc_position='fixed')
```

如我现在需要定义一个名为 Charlie 的角色，其使用的图像名为 charlie，存储角色名的变量名叫 `c_name`：

```
1 define c_name = "Charlie"
2 define c = DynamicCharacter('c_name', image='charlie', what_prefix='"',
    what_suffix='"', ctc="ctc", ctc_position="fixed")
```

现在，我们就成功定义了 Charlie 这个角色。那么这段代码应该放在哪里呢？答案是 game 目录下的 definitions.rpy 文件内。这个文件里储存着所有在游戏中需要用到的资源的定义。

打开编辑器的查找功能，搜索：“`define s = DynamicCharacter`”，随后在这一行上面或下面增添例如上述的代码。

现在，definitions.rpy 文件内的代码应该长这样：

```

1 # 角色变量
2
3 define narrator = Character(ctc="ctc", ctc_position="fixed")
4 define mc = DynamicCharacter('player', what_prefix="", what_suffix="",
5     ctc="ctc", ctc_position="fixed")
6 define s = DynamicCharacter('s_name', image='sayori', what_prefix="",
7     what_suffix="", ctc="ctc", ctc_position="fixed")
8 define m = DynamicCharacter('m_name', image='monika', what_prefix="",
9     what_suffix="", ctc="ctc", ctc_position="fixed")
10 define n = DynamicCharacter('n_name', image='natsuki', what_prefix="",
11     what_suffix="", ctc="ctc", ctc_position="fixed")
12 define y = DynamicCharacter('y_name', image='yuri', what_prefix="",
13     what_suffix="", ctc="ctc", ctc_position="fixed")
14 define ny = Character('夏树 & 优里', what_prefix="", what_suffix="", ctc
15     ="ctc", ctc_position="fixed")
16 define c = DynamicCharacter('c_name', image='charlie', what_prefix="",
17     what_suffix="", ctc="ctc", ctc_position="fixed")
18
19 # ...
20
21 # Default Name Variables
22 default s_name = "纱世里"
23 default m_name = "莫妮卡"
24 default n_name = "夏树"
25 default y_name = "优里"
26 default c_name = "Charlie"

```

4.1.2 增加、定义图片

现在，在 mod_assets 文件夹下创造一个名为 images 的文件夹。在后续的教程中，我们将会把所有的图片资源都存储在这个 images 文件夹内。

警告

请注意，对于所有图片，其格式都应为 **PNG**，且背景应该是透明的，图像编号不能和已有的重复。角色立绘资源的分辨率应为 **960x960** 以保证兼容性，且对于一个完整的立绘应遵循原版 **DDLC** 的原则分成三个图像：头部、左半身、右半身。背景图像的尺寸应为 **16:9**，分辨率应该为 **1280x720**。如果图像资源分辨率过大，您可以使用软件降低图像分辨率或使用 **size** 属性解决：

```
1 size (1280,720) # 添加 size 属性
```

为了兼容性，我们建议您新建的所有文件（夹）名称全部为英文小写字母，且不使用中文。

4.1.2.1 角色立绘

在获取角色立绘后，在 images 文件夹下创建一个文件夹，名为您想要添加立绘的角色名，如 sayori。将图像放进您刚才创建的文件夹内，同时我们建议您将图像名改为 26 个小写字母

或 0-9 数字中的任意一个（建议从 a 开始到 z 依次排列并遵循2.2.2.1中的定义）。

增添图像后，接下来我们就该定义角色立绘了。定义图像的语法如下：

```
1 image <图像名> = "<资源地址>"
```

使用 image 定义后的图像就可以被 show 语句使用了。

由于 DDLC 将角色立绘拆分成为了三部分（请参考2.2.2.1），故对于角色立绘的定义会非常的复杂。

您可以按照以下步骤判断选取相应的代码进行修改：

1. 您添加的立绘是 DDLC 原有角色：

(a) 您只添加了表情：

```
image <角色名> <立绘编号> = im.Composite((960, 960), (0, 0), "<角色名>/<数字>l.png", (0, 0), "<角色名>/<数字>r.png", (0, 0), "mod_assets/images/<角色名>/<图像资源>.png")
```

(b) 您只添加了身体姿势：

```
image <角色名> <立绘编号> = im.Composite((960, 960), (0, 0), "mod_assets/images/<角色名>/<数字>l.png", (0, 0), "mod_assets/images/<角色名>/<数字>r.png", (0, 0), "<角色名>/<字母>l.png")
```

(c) 您同时添加了表情与身体姿势：

```
image <角色名> <立绘编号> = im.Composite((960, 960), (0, 0), "mod_assets/images/<角色名>/<数字>l.png", (0, 0), "mod_assets/images/<角色名>/<数字>r.png", (0, 0), "mod_assets/images/<角色名>/<图像资源>.png")
```

2. 您添加的立绘是新定义的角色：

```
image <角色名> <立绘编号> = im.Composite((960, 960), (0, 0), "mod_assets/images/<角色名>/<数字>l.png", (0, 0), "mod_assets/images/<角色名>/<数字>r.png", (0, 0), "mod_assets/images/<角色名>/<图像资源>.png")
```

例如现在为尤里增添了新的服装与表情，且图像资源名分别为 1l, 1r, happy.png 那么您使用的代码应该长这个样子：

```
1 # m means mod
2 images yuri m1a = im.Composite((960, 960), (0, 0), "mod_assets/images/yuri/1l.png", (0, 0), "mod_assets/images/yuri/1r.png", (0, 0), "mod_assets/images/yuri/happy.png")
```

对于角色立绘的列举，您可以前往 <https://docs.dokimod.top/pages/0a59cf/> 查看。

4.1.2.2 背景图像

在获取背景图像后，在 images 文件夹下创建一个名为 bg 文件夹。将背景放进您刚才创建的文件夹内，同时我们建议您将图像名改为小写的单词。

增添背景后，接下来我们就该定义背景了。定义背景的语法如下：

```
1 image bg <背景名> = "mod_assets/images/bg/<背景图像资源名>.png"
```

扩展知识

这里的 **bg** 指 **background**, 它实际上是图像名的一部分, 其目的是为了区分角色立绘与背景立绘。

4.1.3 音频资源

现在, 在 **mod_assets** 文件夹下创造一个名为 **audio** 的文件夹。将音频放进您刚才创建的文件夹内, 同时我们建议您将音频名改为小写的单词。

警告

Ren'Py 只支持以下几种音频格式:

- Opus
- Ogg
- MP3
- WAV
- FLAC

然后, 打开 **definitions.rpy** 文件, 新增以下代码:

```
1 define audio.<音频名> = "mod_assets/audio/<音频文件名>.<后缀名>"
```

请注意, 音频名不能和已有的定义重复, 否则会覆盖定义。

4.1.3.1 节选播放

Ren'Py 支持将音频切割成一段播放。其中, 共有三种行为:

- from
- to
- loop

from to from 与 to 标签用于指定播放文件的起始位置和终止位置。其基本用法为:

```
1 <from [开始] to [结束]>
```

例如, 现有一个音频名为 “festival.ogg” 存放在 **mod_assets** 文件夹中。现在我们想要从这个音频的第 5 秒开始播放, 到第 15 秒停止, 我们可以修改音频的定义为:

```
1 define festival = "<from 5 to 15>mod_assets/festival.ogg"
```

这样, 在播放 **festival** 音频时就会从第 5 秒开始, 到第 15 秒结束。

loop loop 标签用于指定音频循环播放。其基本用法为:

```
1 <loop [开始, 默认为 0]>
```

例如, 现有一个音频名为 “music.ogg”, 我们想要从第 10 秒开始播放, 并循环播放这个音频, 可以修改音频的定义为:

```
1 define bg_music = "<loop 10>mod_assets/music.ogg"
```



5. 特殊效果及特殊脚本

本节目标包括：

- 了解 DDLC 中一些二周目的特殊效果；
- 了解 DDLC 中的一些特殊脚本；
- 学会定义一首新诗歌并对其进行调用。

在 Mod 中，二周目的一些特殊效果往往是有用的，特别是一些解谜类或恐怖类的 Mod。在本章中，我们将了解 DDLC 模板中的一些特殊脚本与 DDLC 二周目中的一些特殊效果。同时，我们也会在本章学习如何定义一首新的诗歌并调用。

5.1 特殊脚本

打开 game 文件夹，我们能够发现许多 rpy 文件。在本节中，我们将会了解这些文件内所定义的内容、它们的作用与使用方法。

5.1.1 bsod.rpy

bsod.rpy 中定义了 DDLC 二周目中“电脑蓝屏”的特殊效果。代码会自动检测当前电脑的类型、版本，并生成对应的“死机”效果。

5.1.2 cgs.rpy

这个文件中定义了 DDLC 中的所有 CG 内容。

5.1.3 console.rpy

这个文件中定义了假结局中莫妮卡删除 CG 时使用的控制台。其使用方法为：

```
1 call updateconsole(text=<命令>, history=<输出>) # 第一次调用
2
3 call updateconsolehistory(text=<输出>) # 更新输出
4
5 call hideconsole # 隐藏控制台
```

例如：

```
1 call updateconsole(text="print('Hello, World'), history='Hello, World")
```

运行代码，我们发现在游戏的左上角出现了控制台，并自动输入了“print('Hello, World')”这串代码，执行结果为“Hello, World”

5.1.4 glitchtext.rpy

glitchtext.rpy 中定义了乱码文字。我们可以通过调用 glitchtext 函数生成一段乱码文字。其使用方法如下：

```
1 $ <变量名> = glitchtext(<长度>)
```

例如我想要生成一串长度为 200 的乱码字符，并把它赋值给一个名为 gtext 的变量，可以用以下的代码：

```
1 $ gtext = glitchtext(200)
```

5.1.5 poems_special.rpy\poems-tl.rpy\poems.rpy

这三个文件共同存储了在游戏中出现的诗歌。其中 poems_special.rpy 定义了二周目的特殊诗歌，poems-tl.rpy 存储着中文版本的诗歌，poems.rpy 则是原版的诗歌。

我们可以按照以下模板定义一首新的诗歌：

```
1 poem_<编号> = Poem(
2     author = <作者名>,
3     title = <标题>,
4     text = """\
5 <具体诗歌内容>"""
6 )
```

同时我们也可以通过 showpoem 来展示诗歌。如下例：

```
1 poem_a = Poem(
2     author = "sayori",
3     title = "Empty",
4     text = """\
5 婚姻，热情，童年，快乐，色彩，希望，友人，可爱，松软，单纯，糖果，购物，狗狗，
6 猫咪，云朵，决意，自杀，想象，秘密，活泼，存在，璀璨，绯红，飓风"""
7 )
8 call showpoem(poem_a)
```

5.2 特殊效果

在本节中，我们将学习 DDLC 二周目的一些特殊效果。

5.2.1 撕裂效果

撕裂效果是 DDLC 在二周目中最常用的一个错误特效，在默认情况下，撕裂效果会把屏幕撕裂成 10 份，并产生混乱错乱的效果。它被定义在 effects.rpy 文件中。

撕裂效果参数的定义如下：

```
1 screen tear(number=<整数>, ontimeMult=<数字>, offtimeMult=<数字>, offsetMin=<数字>, offsetMax=<数字>, srf=<Surface对象>)
```

其中, number 参数对应的会将屏幕等分成多个部分, 默认为 10; ontimeMult 与 offtimeMult 作为撕裂部分来回摆动的时间的乘积因子, 默认为 1; offsetMin 与 offsetMax 设置了偏移值的最小值与最大值, 默认为 0 与 50; srf 要求提供一个 Surface 对象 (可以理解成将当前屏幕截图), 当参数为 None 时, 会自动生成一个 Surface 对象, 默认为 None。

一般的, 我们可以直接使用下列代码做到最完美的撕裂错误效果:

```
1 show screen tear(20, 0.1, 0.1, 0, 40)
2 play sound "sfx/s_kill_glitch1.ogg"
3 $ pause(0.25)
4 hide screen tear
```

5.2.2 死机效果

在上一章节中, 我们学习到了 bsod.rpy。bsod 的参数定义如下:

```
1 call screen bsod(bsodCode=<字符串>, bsodFile=<字符串>, rsod=<bool值>, chinese_screen=<bool值>)
```

其中, bsodCode 是一串伪造的错误代码, 默认为 DDLC_ESCAPE_PLAN_FAILED; bsodFile 是导致蓝屏的文件名, 默认为 libGLESv2.dll。rsod 表示是否在将“蓝屏”替换为“红屏”, 默认为 False; chinese_screen 表示是否在使用中文版本, 默认为 True。

例如:

```
1 call screen bsod()
2 # 或者
3 call screen bsod("CRITICAL_PROCESS_DIED", "DDLC.exe")
```

5.2.3 对话修改效果

在二周目中, 我们会经常看到经莫妮卡修改过的对话。要使用这种效果只需要在对话前加入以下代码:

```
1 $ style.say_dialogue = style.edited
```

扩展知识

`style.say_dialogue` 是一个常量, 决定了对话的样式。在后面学习了样式的定义后, 您甚至可以自己编写一个样式代替原版。

若要恢复到正常文本, 只需要添加:

```
1 $ style.say_dialogue = style.normal
```

如下例子:

```
1 $ style.say_dialogue = style.edited
2 "你爱我吗?"
3 $ style.say_dialogue = style.normal
4 "...我这是怎么了?"
```

5.2.4 画面心跳效果

在二周目中，优里与夏树在第二天争吵时，屏幕会出现心跳效果。要使用这段效果，请在代码中添加：

```

1 $ timeleft = 12.453 - get_pos() # 如此莫名其妙的一段代码。
2 show noise zorder 3 at noisefade(25 + timeleft)
3 show vignette as flicker zorder 4 at vignetteflicker(timeleft)
4 show vignette zorder 4 at vignettefade(timeleft)
5 show layer master at layerflicker(timeleft)

```

第二行代码会在屏幕上显示噪点。

第三行到第五行代码会让屏幕的四个角落暗淡并晃动。

5.2.5 操控历史对话

在二周目中，有时候优里与玩家的一些异常对话无法在历史里找到或者与玩家在游玩过程中看到的内容不一样。要做到这种效果，我们可以对 `_history_list` 进行修改达到上述效果。

要删除对话，我们可以使用 `pop` 函数删除。如下例：

```

1 m "我正在想..."
2 m "你希望我来到你的世界吗，[player]? {nw}"
3 $ _history_list.pop()
4 m "你想要吃小蛋糕吗？"

```

在游玩过程中，玩家会正常看到“你希望我来到你的世界吗”这句话，但是在历史对话中，玩家看到的却是：

```

1 莫妮卡 "我在想..."
2 莫妮卡 "你想要吃小蛋糕吗？"

```

要修改对话，我们可以利用索引修改历史对话。如下例：

```

1 s "... "
2 s "救救我，[player]! {nw}"
3 $ _history_list[-1] = "... "

```

在历史对话中，玩家只能看到：

```

1 纱世里 "..."
2 纱世里 "..."

```



6. 分发 Mod

本节目标包括：

- 修改 Mod 的设置；
- 初步学习分发你的 Mod。

编写完模组后，为了保护代码与方便传递模组，我们需要将我们的代码封装成为 rpa 文件。但在打包前，我们需要修改一些设置，保证不与其他同样使用 DDLC 中文模板的模组冲突。

6.1 修改设置

在 game 文件夹下，有一个 options.rpy 文件，在其中定义了一些与 Mod 个性化的一些设置。在打包您的模组前，我们需要修改一些设置来保证模组不会与其他模组冲突。

现在，打开 options.rpy 文件。找到以下代码：

```
1 define config.name = "DDLC 中文 Mod 模板"
2 define config.version = "2.0.0-dev"
3 define gui.about = _("""这里是写简介的地方。在 options.rpy 里写上你的 Mod
   简介吧!""")
4 define build.name = "DDLCModTempCNNext"
5 define config.save_directory = "DDLCModTempCNNext"
```

按照以下内容修改上述代码：

- 将 config.name 的内容修改为您的模组的名字；
- 将 config.version 修改为“1.0.0”（您也可以自定义版本号）；
- 将 gui.about 修改为您模组的简介；
- 将 build.name 修改为您模组的名字；

必须注意

请注意只能使用英文字母，中文、下划线等会导致打包失败。

5. 将 config.save_directory 修改为您模组的名字。

必须注意

请注意只能使用英文字母，中文、下划线等会导致游戏无法打开、持久化数据无法保存等问题。

如：

```
1 define config.name = "Monika 大战 360" #???
2 define config.version = "1.0.0"
3 define gui.about = _(""""Monika想要删除文件，可是万恶的360却拦住了她。接下
   来会发生什么呢...""")
4 define build.name = "MonikaFightWithThreeSixZero"
5 define config.save_directory = "MonikaFightWithThreeSixZero"
```

然后，打开 definitions.rpy 文件，找到以下代码：

```
1 define persistent.demo = False
2 define config.developer = False
```

确保上述两个常量为 False。

完成后，您现在可以打包您的模组了。

6.2 打包模组

现在，打开 Ren'Py Launcher，点击您的项目，在右边您应该可以找到一个选项名为“构建发行版”。点击后，Ren'Py 会自动扫描一遍您的项目。

扫描完成后，您会见到一个有许多选项的界面。首先，确保“选项”中的“向 call 语句添加 from 从句，执行一次”勾选了。

警告

call 语句在没有 from 从句的情况下，会有存档损坏的风险。

然后，取消“构建分发包”中的“PC: Windows and Linux”与“Macintosh”，勾选“Ren'Py 7 DDLC Compliant Mod”，点击“构建”按钮。稍等片刻，您的模组就会被 Ren'Py 编译打包成压缩包了。

警告

请注意，在扫描项目与编译打包过程中，Ren'Py 可能会“假死”。假死的时间取决于您电脑的配置与您模组的大小。在假死期间，不要强制关闭 Ren'Py，您只需要静待其编译完成后自动恢复。若 Ren'Py 被强制关闭，其生成的模组很有可能是损坏的，需要重新生成。

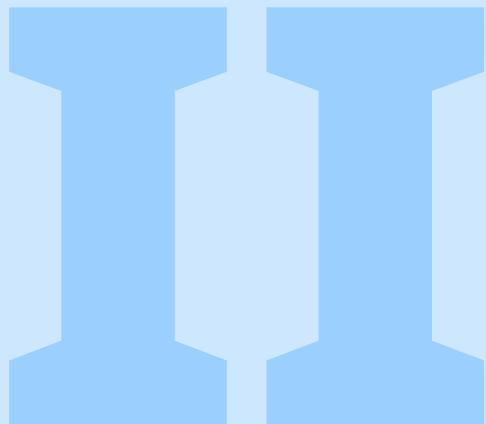
打包完成后，Ren'Py 会自动打开生成的模组的位置。打开压缩包，确保在其 game/文件夹下没有 audio.rpa、fonts.rpa 与 images.rpa 后，您便可以上传您的模组共其他玩家游玩了。

必须注意

根据 Team Salvato 的 IP Guidelines，您应该注意以下几件事：

1. 模组里禁止（MUST NOT）包含 DDLC 原版内容；
2. 您不能（NOT ALLOWED）将您的模组上传到任何一个应用商店；
3. 您不能（MAY NOT）将 DDLC、DDLC Plus、您的模组移植到其他平台（事实上，现在有很多的模组都可以在手机上游玩，这原则上违反了 IP Guidelines，但是 Dan 鸽睁一只眼闭一只眼也就过了）；
4. 您的模组不能（NOT ALLOWED）设计在 DDLC 之前游玩；
5. 您的模组必须（MUST）是免费的，且在游戏内不能（MAY NOT）包含任何付款或捐赠链接，也不得鼓励玩家在游戏中捐赠、购买商品或捐款。

移植模组到手机平台无疑能够扩大 DDLC 社区。但是若要把模组移植到手机平台，在目前就不得不包含 DDLC 原版内容。包含 DDLC 原版内容与移植本身是违反 IP Guidelines 的（但本书依然会介绍如何将您的模组移植到手机）。Dan 鸽虽然睁一只眼闭一只眼，但这并不意味着我们就能肆意违反 IP Guidelines。若您的模组没有移植计划，那么您就不应该制作所谓的“解压即玩”、“傻瓜包”版（即包含原版 DDLC 内容）。请记住：保护知识产权不仅是保护作者的权利，更是在促进 DDLC 社区的良好发展。



进阶部分

第七章 更新 Ren'Py	61
7.1 判断您是否需要升级	61
7.2 更新 Ren'Py 版本与模板版本	62
第八章 自定义 GUI 与定义新界面	64
8.1 Ren'Py 的界面显示机制	64
8.2 自定义 GUI	65
8.3 界面语言	65



7. 更新 Ren'Py

本节目标包括：

- 更新你的 Ren'Py 版本；
- 学会解除兼容性警告。

首先，恭喜您完成了基础篇的学习，来到了进阶篇。在进入正文以前，我们不得不提醒您需要有一定的 Python 基础才能完成进阶篇的学习。仅靠本书浅显的介绍是远远不够的，因此，我们建议您在进行一段时间的模组开发和 Python 学习后，再来阅读本部分。

本章我们将会学习如何将您的模板更新为 4.0 版本或 5.0 版本以启用更多功能，并使用 Ren'Py 8 进行进一步的开发。

警告

请注意，目前 DDLC 中文模板已经有 2 年未更新，最新版本的 Ren'Py 不一定能够完美适配模板。若 Ren'Py 8 后续版本进行了不兼容的修改，可能会让模板运行错误。因此我们推荐您使用最后一个确定适配的版本——Ren'Py 8.0.3。但最新的 Ren'Py 8 也带来了一些新特性，为了使用最新版本的 Ren'Py 8，我们需要解除兼容性警告。总而言之，本书会向您介绍如何使用最新版本的 Ren'Py 8，但本书中的代码只保证能在 Ren'Py 8.0.3 上正常运行，不保证最新版 Ren'Py 8 能够成功运行。

7.1 判断您是否需要升级

首先，升级模板是一个很复杂的过程。尤其是将模板从 Next 分支（Python 2 版本）升级到 Future 分支（Python 3）这种跨大版本的升级。一般来说，您的代码仅需较少的改动或完全不需改动便可迁移到 Future 分支。但由于 Python 2 与 Python 3、Ren'Py 7 与 Ren'Py 8 之间的差异较大，相同的代码在不同版本上运行可能会有不同的效果。为了判断您是否需要升级，本部分将会给您参考建议，帮助您判断是否需要升级。请按照以下步骤进行判断：

1. 您是否需要或想要使用 Ren'Py 8 或 Python 3 的一些新特性（如多线程操作、格式化字符串、Keyword-Only Arguments、f-string、关键词参数等）

2. 您是否有足够的技术修正代码预期外的运行效果
3. 您能否接受升级到 Future 分支所可能带来的游戏的不稳定

若以上三个条件均满足，我们建议您升级到 Future 分支。您也可以出于尝试目的升级您的项目，只要先进行备份。

7.2 更新 Ren'Py 版本与模板版本

在一切开始之前，请先对您现在的项目文件夹进行备份，只需将您的项目文件夹压缩成压缩包或复制一份即可。

警告

截止到本章编写时，Ren'Py 7 的支持即将结束，故我们建议您直接升级至 Future 分支与 Ren'Py 8 版本，而不是停留在 Ren'Py 7。

7.2.1 更新 Ren'Py

更新 Ren'Py 版本是一个相对简单的步骤。首先，前往<https://www.renpy.org/latest.html> 下载 Ren'Py 8 的最新版本。

然后，按照第1.2节完成对 Ren'Py 的配置。

7.2.2 更新模板

现在，前往<https://github.com/DokiMod/DDLCModTemplate-Chinese-future>，按照第2.1.1节的步骤，下载 Future 分支并完成配置。

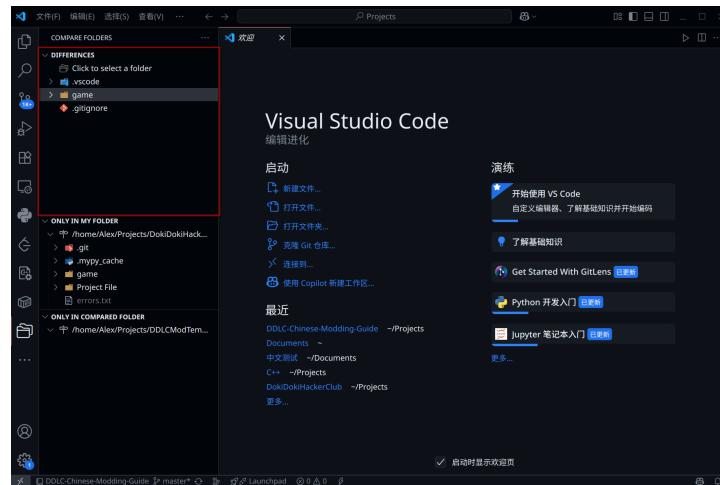
接下来，我们将进入最复杂的一个环节——迁移修改过的代码。首先，打开 VS Code，按照第2.1.2节步骤安装 Compare Folders 插件。随后，使用 VS Code 打开您项目文件夹所处的文件。例如：我的项目名字叫作“ClubStory”，储存在名为“项目”的文件夹，那么我应该使用 VS Code 打开“项目”文件夹而非“ClubStory”文件夹。

然后，按照第2.1.1节的步骤，下载 Next 分支，放入您项目所在的文件夹中（按照上例，您应该放入“项目”文件夹而非“ClubStory”文件夹）。在 VS Code 中，单击选中 Next 分支文件夹，再按住 Ctrl 键（Windows、Linux）或 Command 键（macOS）选中您的项目，右键选择 “[Compare Folders] Compare selected folders”。等待一段时间后，您应该看到如图所示的界面。在跳出的选项卡中，我们只需关注“DIFFERENCES”和“ONLY IN COMPARED FOLDER”两栏中的 game 文件夹。

在 VS Code 的文件选项卡中，选择新建窗口，在新的窗口中打开 Future 分支。回到原来的窗口，聚焦于“DIFFERENCES”栏目中，展开 game 文件夹，依次点击每个文件夹下的文件，查看文件的差异，并将修改过的代码同步修改到 Future 分支的对应文件下。

然后，聚焦于“ONLY IN COMPARED FOLDER”栏目中，这里是您添加的文件，将它们全部复制到 Future 项目中的对应位置。

到此，我们就完成了代码的迁移，接下来就是使用 Ren'Py 8 运行项目进行一轮完整的游戏，检测是否有代码运行结果在预期之外，并进行修复。



7.2.3 解除兼容性警告

初次运行项目，若您使用的 Ren'Py 版本高于 8.0.3，您可能会遇到以下提示：

警告：您目前用于开发 DDLC 模组的 Ren'Py SDK 未经过模组兼容性测试。

目前最新的适用于 DDLC 模组的 Ren'Py 8 版本为<https://www.renpy.org/releases/8.0.3> “Ren'Py 8.0.3”。

在过高版本的 Ren'Py 上运行 DDLC 或其模组可能会引发问题，也可能导致游戏功能损坏。

这个提示将会在初次运行时出现，原则上当您按照本书的打包教程，将 config.developer 设置为 False，那么这个提示将不会被展示。但以防万一，我们仍建议您从根源上取消该提示。

现在，关闭游戏，打开 game/core/lockdown_check.rpy 文件，在 label 语句后添加一行，输入以下代码（注意缩进）：

```
1     return
```

再次打开游戏，您会发现这段提示永久消失了。



8. 自定义 GUI 与定义新界面

本节目标包括：

- 了解一些 Ren'Py 中的特殊界面；
- 学会对原版的 GUI 进行自定义，修改颜色、背景、字体等；
- 学习定义界面与使用界面语言

GUI (**G**raphical **U**ser **I**nterface) 即图形用户界面。您在游戏中所看到的主界面、对话框、选项、设置界面等都属于 GUI。也许您已经厌倦了原版 DDLC 的粉红色界面，或是想要将界面颜色改为符合剧情内容的颜色，想要添加一些新的界面，例如一个日历界面，通过本章的学习，您将能够实现自定义您的主界面、对话框等。

8.1 Ren'Py 的界面显示机制

玩家在 Ren'Py 游戏中所见到的一切，都可以大致分为两部分：图像 (Image) 与用户接口 (User Interface)。用户接口包括但不限于：对话框、对话角色名、设置菜单、游戏的主界面等。通过基础部分的学习，我们已经学会了使用 scene、show 和 hide 语句向玩家展示图像。界面就是 Ren'Py 为开发者所提供的一种定义用户接口的方式。在本节中，您将会了解 Ren'Py 的界面显示机制与初步学习界面语言。

界面主要有两个功能：第一，向用户显示信息。信息的可以是文字、图像等。例如，say 界面用于向玩家展示对话，包括人物和内容两大部分。第二，允许玩家与游戏交互，例如：按钮。这种界面允许玩家触发某些行为，例如保存游戏与读取存档。

界面主要由组件组成。组件包括：框架 (Frame)、窗口 (Window)、文本 (Text)、按钮 (Button)、纵向排列组件 (Vbox) 等。这些组件可以通过用户接口语句于界面语言中定义。

一般来说，界面将会在以下情况被显式调用或隐式调用：

- 通过 “show screen” 等语句被调用；
- Ren'Py 底层机制自动调用，如游戏启动时，Ren'Py 会自动调用 main_menu 界面；
- 部分代码隐式调用，如使用 menu 语句时，会自动调用 choice 界面；
- 将用户的操作与界面绑定，如按下 ESC 时，调出 navigation 界面；

同时用户在界面上的每次操作（包括移动鼠标），都会使界面刷新。

8.2 自定义 GUI

8.3 界面语言

界面语言是定义一个用户接口最正式的办法。它包含定义新界面的语句（screen 语句）、添加可视组件至界面的语句和控制型语句。

8.3.1 初步了解界面语言的结构

以模板内定义的 say 界面为例：

```

1 screen say(who, what):
2     style_prefix "say"
3
4     window:
5         id "window"
6
7         text what id "what"
8
9         if who is not None:
10
11             window:
12                 style "namebox"
13                 text who id "who"
14
15 # 如果有对话框头像，会将其显示在文本之上。请不要在手机界面下显示这个，因为
16 # 没有空间。
17 if not renpy.variant("small"):
18     add SideImage() xalign 0.0 yalign 1.0
19
20 use quick_menu

```

首先，第 1 行的“screen say(who, what):”定义了一个名为 say 的界面，并且该界面在调用时接受两个参数：who 与 what。screen 是最基础也是最重要的一个界面语言语句，用于定义新界面。screen 语句接受一个参数，即界面名，同时可以在界面名后用英文括号定义这个界面所接受的参数。转换为代码如下形式：

```

1 # 接受参数
2 screen <界面名>(<参数1>, <参数2>, ...):
3 ...
4 # 不接受参数
5 screen <界面名>:
6 ...

```

第 2 行的“style_prefix ”say”，是 screen 语句的一个特性，包括特性名称和特性值，特性值是简单表达式。“style_prefix”是特性名称，“”say””则是特性值。该语句表示在该界面内的所有组件的样式（style），都以“say”为前缀。例如，第七行的“text”组件，将会默认使用“say_text”这个样式，除非通过 style 参数重新指定一个样式。

扩展知识

事实上，该组件所用样式应为 `say_text`，但由于修改组件 `id` 特性为“`what`”，Ren'Py 内部使用 `say_dialogue` 替代 `say_text`。无论如何请记住，`style_prefix` 效果如它的字面意思一样。

第 4、7、11、13、18 行，分别向这个界面中添加了不同的组件。由此可见，大多数界面语言语句使用通用的语法。语句的开头以某个关键词进入。如果语句使用参数，参数就跟着开头的关键词后面，参数列表是使用空格分隔的简单表达式。同时，后面一般会跟一个特性（property）列表，多个同级特性组成一个特性列表，特性列表中各个特性使用空格分隔。例如，第 7 行的 `text` 组件，将参数 `what` 作为 `text` 组件的参数，`id` 则是 `text` 组件的一个特性，“`what`”代表着这是这个特性的值。

第 1、4、11 行，`screen` 语句和 `window` 组件后面跟着一个英文冒号，代表该组件是一个语句块（Block）。多个同一等级的语句组成一个语句块。语句块的内容可以是以下内容：

- 特性列表；
- 界面语言。

例如，第 1 行后的所有组件，都是 `screen` 语句下的代码块；第 7 行的 `text` 组件，是在 `window` 组件下的语句块，可以理解为 `text` 组件被包含在 `window` 组件内。