Python Basics

李轶凡 科协网络部

参考资料: Docs9 Python 基础； ayf 的 2021 年暑培讲义; zcy 的 2022 年暑培讲义

---

1 引入

1.1 Python 的历史

Python 是一种高级编程语言，它具有简单易学、代码可读性强、功能强大等特点，被广泛用于数据科学、Web开发、人工智能等领域。相应大家也经常从 bilibili、微信朋友圈等各种地方等牛皮癣小广告中看到 Python 的身影，这门语言目前已稳居 TIOBE 排行榜榜首，是当之无愧的最热门的编程语言。

Python 的历史可以追溯到1989年，由荷兰计算机科学家 Guido van Rossum 在圣诞节假期时开始开发一种新的编程语言。他最初的想法是创建一种易于阅读、易于学习的语言，同时具有与 C 语言一样的能力。在经过多年的发展和完善后，Python 成为了一种非常受欢迎的编程语言。2000年，Python 2.0 发布；Python 2.x 系列一直持续到 2020 年，发布了 Python 2.7.18 版本作为最终版本。

2008年，Python 3.0 发布，引入了一些不兼容的改变，如 print变成了 print() 、除法运算符 / 的行为改变等。这些改变是为了解决 Python 2.x 系列中存在的一些设计缺陷。Python 3.x 系列逐渐成为主流版本，目前最新的版本是 Python 3.11。

大家可以看到，Python 2 和 Python 3 的开发周期是有重叠的，而且 **特别要强调** 这两个版本的 Python 是不兼容的，这个和大家之前学习的 C / C++ 向下兼容不太一样。

1.2 Python 的特点与优势

为什么 Python 会这么热门呢？大致有以下这些原因：

1. 简单易学：Python 语法简单，容易上手；代码可读性好，结构清晰，因此，Python 常常作为非计算机专业的人员所使用的编程语言，可以方便、快捷地协助完成其他领域的科研工作；同时，也作为编程启蒙的语言存在。

2. 解释型语言：Python 是一种解释型语言，可以直接运行代码，无需编译。这使得 Python 的开发和调试过程更加快速和方便。大家目前可能没有体会，但是当你遇到了一些比较大的项目的时候，编译就会是一个耗时极长的过程，而且如果出现了错误，很可能整个程序直接崩溃，也不方便于定位错误；解释形语言在这方面就好得多。

   > 解释型语言每次执行程序都需要一边转换一边执行，用到哪些源代码就将哪些源代码转换成机器码，用不到的不进行任何处理。每次执行程序时可能使用不同的功能，这个时候需要转换的源代码也不一样。相比于编译型语言，解释型语言几乎都能跨平台，“**一次编写，到处运行**”是真实存在的。也就是说，可以轻松实现在 Windows 上编写 Python 源代码，然后丢到 Linux 上去运行，只要双方配置了相同的解释器环境即可。

3. 强大的库：Python 自带了很多标准模块和库，如 re（正则表达式）、os（操作系统接口）等；同时，Python 社区庞大活跃，有很多优秀的第三方库和框架，如 NumPy、pandas、Django、Flask 等，这些库可以帮助程序员更快速、更方便地实现各种功能，提高开发效率。例如，想要实现一个最基本的网页服务器，Django 只需要半小时，而你用 C 来写....emmm

这些因素的共同作用下，Python 有一个极大的优势：开发效率高，也就是节约我们，作为程序员，写代码的时间；与之相对的，Python 还有一个极大的劣势：运行效率低。如果你像我一样是一个算法竞赛选手，你可能会特别不理解...

除此之外，还有一些别的原因：

4. 跨平台性：Python 可以在 Windows、Linux、macOS 等各种操作系统上运行，且代码具有高度的可移植性。这使得 Python 成为跨平台开发的首选语言之一。不像是 C++ ，一次编译出来的代码只能在当前平台运行，如果要跨平台交叉编译会是件很困难的事情；由于解释运行的特性，Python 代码的兼容性很好。

等等.

2 Python 初印象

在这一部分，我们将会对照着 C++，将 Python 一些最基础的语法规则交给大家，这一章节后大家或许就可以把原先写过的 C++ 代码“翻译”成 Python 代码了（笑）

2.1 运行

我们刚刚提到了 Python 是一个解释型语言，因此在安装完成后，大家就可以在终端输入 Python ，进入 **交互式窗口** ，一行一行的执行代码；如果这一行有返回值的话，也会直接输出出来。

上文提到的“输出”，除非是 print 这样的明确输出函数，否则一般情况下它会将执行语句的返回值转化成表示串，这和 print 的效果或许不同，后面讲字符串部分时会举例子。

> 退出界面的方法是按 `Ctrl+D` 或者执行 `exit()` （Windows 用户请使用 `Ctrl+Z` ）。

当然，我们也可以把需要运行的 Python 脚本写在文件里，然后在终端输入 `python <your file>` 就可以运行了，这和大家跑 C++ 比较相似；不过大家目前还不着急安装 Python，因为后面的讲解中会涉及一些和 Python 环境管理、交互有关的内容。如果目前还没有安装，在上课时，我推荐先去一些线上的平台实验，例如说 programiz 和 w3cschool。

2.2 输出

Python 的输出十分简单明了：

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

假设我们已经定义了变量 a, b，我们可以这样使用 print

```
print("变量是", a, b)
```

注意到，这些不同元素之间会有空格，结尾会有换行符。怎么样自定义格式呢？后续的章节中会涉及。

2.3 变量定义

Python 中的变量类型与运算和 C++ 中大致相同，有 `int`，`float`，`bool`，`NoneType` 这些，具体细节将在下一章中介绍，这里我们先看怎么定义一个变量：

```
a = 124
b = "tsinghua"
c = None
```

注意到不需要变量定义关键词，也不需要声明变量类型，我们可以直接使用变量。不过，如果你尝试对一个未声明的变量作为右值/进行运算，就会得到报错。

> 我们尝试与熟悉的 C++ 语言对比，解释静态类型语言与动态类型语言的区别。
>
> **静态类型语言**中，每个变量都有确定的数据类型。变量的类型在编译期就被完全确定，且此后变量的类型无法改变。
>
> 与其相对应的是**动态类型语言**：变量并没有确定的数据类型，这意味着变量的类型可以在运行期改变。
>
> 比如，在 C++ 中：如果我们定义了 string 类型的 a 变量，在同一个作用域内我们不能尝试使用 `a = 1`，因为我们已经规定了 `a` 是 `std::string` 类型。
>
> 而在 Python 中，我们可以尝试：
>
> ```
> a = 123
> a = 100.0
> a = "string"
> a = [1,2,3] # List, would be explained later
> ```
>
> 我们并没有对变量 `a` 进行过声明，而可以直接赋值使用，并且在使用过程中可以随意地变换其类型。
>
> Reference: https://blog.csdn.net/alashan007/article/details/100742591.

同时，Python 还是强类型语言，也就是说，不同类型的值之间不可以隐式地相互转换。例如，下面这一段代码是不能运行的：

```
a = "987"
b = 320
print(a + b)
```

当然，手动转换类型不会遇到任何阻碍：

```
print(a + str(b))
```

2.4 输入

Python 里面的输入也很简单：

```
age = input("请输入你的年龄")
```

不过这样做默认得到的是字符串。如果你想要得到数字的话：

```
age = int(input("请输入你的年龄"))
```

即可。当输入的字符无法转化为 int 的时候，会抛出一个 ValueError 异常

2.5 运算

Python 中大部分运算符和 C++ 是一致的，在这里就不做赘述了，需要注意的有下面几个：

- `/` 是真除运算符，会得到浮点结果
- `//` 是整除运算符
- `**` 是幂运算符，例如 `3 ** 5` 代表 3^5
- 逻辑运算符 `&&` -> `and`，`||` -> `or`，`!` -> `not`，注意到 对于 *`and`* 或 *`or`*，得到的结果不会直接转换为 *`bool`*，而是能够得出结果的最后一个变量。
- 三目运算符 `a if condition else b`，例如 `x if x is not None else 0`
- 包含运算符 `in` 和反义的 `not in`
- 同地址运算符和不同地址运算符 `is` 和相应的 `is not`

2.5.1 `is` 运算符

按照上述定义，大家可能会把 `is` 理解为 C++ 中的取地址再比较这样的操作，但在 Python 中它的实际应用场景不止是这样。比较常见的是用来判断一个变量是否是 `None`。趁这个机会，给大家介绍一下 Python 中的 `None`：

> 在Python中，`None`是一个【内置的常量】，表示值的缺失或对象的缺失。它是一个特殊值，通常用于指示变量或对象没有值或尚未初始化。
>
> `None`经常用作函数的默认返回值，这些函数不返回任何有意义的值；`None` 也可以指示变量或对象当前未初始化

由于它是一个内置的常量，所有值是 None 类型的变量都指向同一个地址。因此，当我要确认一个变量是否是 `None` 时，可以并且推荐使用 `is` 运算符，而非 `==` 运算符。

大家可以思考一下把一个变量赋值为 `None` 和在 C++ 中将其地址设置为 `NULL` 是一回事情吗?

由于一些奇怪的特性，不止 None 可以由 is 来比较，一些较小的整数也可以用 is 来比较（当然并不推荐这么做）：

```
In [39]: x = 3
In [40]: x is 3

# 输出
# <>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
# <ipython-input-40-43bab69cc802>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
#  x is 3

Out[40]: True
```

> 为什么呢？因为 Python 会存储所有的-5到 256 的整数，其他任何变量是这些值时，会被指向这个预先开好的内存，因此任何两个值为 3 的 int 变量都指向同一内存地址。

2.6 控制流

Python 脚本是可以拎出来一行一行地执行的，因此，即使写在文件里的 Python 代码，也是不需要 main 函数作为入口的（文件开头就是入口）。这种实现可能不太美观，因此我们常常会用 `if __name__ == "__main__":` 作为主函数。

2.6.1 顺序语句

Python 语言推荐的做法是依靠换行来隔离语句，这时候分号不是必须的；你想要加上分号也可以，也可以在一行内写上多个语句，但这不是推荐的做法，也请你不要这么做。

2.6.2 嵌套

在 Python 这一门语言中，是不用大括号 `{}` 表示代码块的；我们使用缩进来表示代码块之间的嵌套关系。如果大家写 C++ 的时候养成了良好的缩进习惯，那么使用 Python 的时候不会对缩进控制代码嵌套有任何不适。Python 中同一层级的嵌套就用同一个缩进表示。

缩进可以是一个 tab 也可以是四个空格，但是在同一份代码文件中必须统一。我们更推荐使用四个空格，像是 VSCode 默认也是使用四个空格。

有一个段子是，据说 Python 程序员工作时都带着一把游标卡尺。

2.6.3 `if`

我们现在来看一下 Python 中的 `if` 语句：

```
if x < 3:
    print("x < 3")
elif x == 4:
    print("x == 4")
else:
    print("x > 4")
```

`else` 和 `elif` 都是可选的；大家可以对比一下和 C++ 的区别 —— 冒号和缩进是必不可少的，条件里的括号可以被省略。 `elif` 其实就是 `else: (换行) if` 的意思，将两个关键词写在一起可以让代码更加简洁，同时也减少很多不必要的缩进。

多层的嵌套会导致左边出现大量的缩进，所幸在 IDE的帮助下这不会为编程带来太大困难，但是仍然观感很差，这种情况下建议使用函数“转移矛盾”。

事实上，任何一种语言多层嵌套总需要缩进，多层嵌套无论如何都是会影响阅读、带来较大的心智负担，我们在实践的过程中要尽量避免这种情况。

2.6.4 `while` 循环

```
a = 10
while 2 <= a < 20: # 语法糖，这种写法是被建议使用的
    print(a)
    a -= 1         # 注意Python中没有自增1和自减1运算符
```

如果你在交互式命令行里写带层次的代码，那么写完最后一个层次后要连按两次回车

2.6.5 `for` 循环

Python 中我们一般不会使用 C++ 中 `for(int i = 0; i < 3; i++)` 这样的循环模式；在大部分情况下，我们会迭代一个可以被迭代的对象，例如列表、元组、字符串：

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

虽然会 C++ 的同学可能有些不适应，但这个其实是很直观的。我们也可以迭代一下字符串：

```
a = 'hello world'
for c in a:
    print(c)
```

注意到 Python 中 for 循环有且只有 `for x in y` 这样一种格式，那想要类似 C++ 这种循环方式该怎么办呢?

```
for i in range(100):    # 暂且理解 range(100) = [0, 1, 2, ..., 99]
    print(i)            # 本质上是在一个可迭代对象中遍历所有元素

# 我们也可以这样使用 range
for _ in range(0, 11, 2):
    print("Hi")
```

同时， `break`, `continue` 这些语句在 Python 的循环当中也是可以正常使用的。

到这里，你就可以说你会写 Python 啦!

3 Python 变量类型

在之前的章节中我们大致提到了 Python 的几种变量类型，现在我们来深入认识一下。

3.1 `int`

Python 中的 int 是 **变长整数**，默认是4字节，有需要时自动增加，也就是说，Python 默认实现了高精度这一功能；除了十进制外还支持十六进制、八进制和二进制的表示。

从这里开始，大家应该显著体会到什么叫做 "Python 写起来比较方便" 了。

3.2 `float`

8 字节浮点数，相当于 C 系语言的 `double`。注意到他们遵循的都是相同的 IEEE 标准，因此会有相同的误差情况， Python 的 float 也不会自动变长。

3.3 `bool`

众所周知的 `True` 与 `False`，注意**首字母大写**，用作数值计算时与 C 系语言一样分别视作 `1` 与 `0`。

3.4 `NoneType`

就像刚刚提到的那样， `None` 其实有一种单独的类型。

3.5 字符串

Python的一个便利点在于将字符串与字节串封装成了基本类型并封装了许多接口与运算。 注意基本类型本身是不可修改的(注意区分赋值与修改)，这与后面的容器不同。

3.5.1 `str` 字符串

`str` 是 Python 中的字符串类型，每个位置存的是一个 Unicode 而非 ASCII，因此原生支持表示汉字。Python中没有单独的字符类型，单个字符将被视为长度为1的字符串。

构造字符串的方法是使用单引号或双引号，没有任何差别的，转义符的使用与C基本一致。不过，当你存储的字符串中可能出现 `"` 或者 `'` 的时候，就可以用另外一者作为字符串标记符。

```
x = '你好'
y = "你好吗"
z = '我很"好'

print(z)
# Output: 我很"好

k = '你不'好'
# SyntaxError: invalid syntax

k = '你不\'好'
print(k)
# Output: 你不'好
```

对于字符串，有几个重要的函数：

- `len(obj)`：`__len__` 常用于获取字符串(注意是 Unicode，因此中文和英文字符一样长)、字节串或容器的长度
- `chr(i)`：将整型变量 `i` 转化为长度为1的的字符(串)
- `ord(c)`:获取(长度为1的)字符(串)的编号(Unicode)

3.5.2 字符串格式化

在 C 中，我们使用 `%d` 等标识符格式化字符串；在 Python 中，我们有一种更加符合直觉的 f - string。我们在字符串的前面加上一个 f, 字符串中通过 `{变量}` 的形式引入变量，这里的变量可以是字符串，也可以是整数等，不需要手动区分。下面是一个例子：

```
a = 333
b = "three"
s = f"{b} {b} {b} is {a}"
```

这种用法在输出的时候尤其常见。

还有两种格式化字符串的方法，由于不是特别常用，这里我就不介绍了。

3.5.3 字节串

`bytes` 是 Python 中的字节串，它表示最纯粹的“二进制”数据，非常像 C 里的 `unsigned char *`，但是在显示上 它仅支持 ASCII 显示，用肉眼看显得有些不伦不类，通常它只存在于数据的处理过程中。

`bytes` 的构造与字符串类似，但是要加一个 `b` 做前导:

```
print(b'123')
# Output: b'123'

print(b'\1')
# Output: b'\x01'
```

```
print(b'\x41')
# Output: b'A'

print(b'\x61\x62')
# Output: b'ab'

print(len(b'\x61\x62'))
# Output: 2
```

其中 `\x` 表示将字符转义成对应的 ASCII 字符。

3.5.4 编码与解码

Unicode 是通用的编码，它在世界范围内提供了统一的字符集编码。传输过程中裸传 Unicode 可能引起空间浪费等问题，因此有了对 Unicode 的不同编码，如 UTF-8、UTF-16、GBK 等等，各有各的特点(例如 GBK 存储中文所需要的空间更少)。它们统称为 Unicode 转换格式 (Unicode Transformation Format)。对于我们日常见到的文件，一个汉字两个字节的常是 GBK 编码，一个汉字三个字节的常是 UTF-8编码。

字符串本身代表的是 Unicode，但是在文件存储时需要指定编码，例如存储为 UTF-8 或 GBK。编码后的字符串就作为 bytes 即字节串而存在了。为了快捷地满足这一需求，`str` 自身有函数 `str.encode` 可以编码成 `bytes`，而 `bytes` 也有函数 `bytes.decode` 可以解码成 `str`。大部分情况下的默认编码是 UTF-8，Windows 在有些场合也会用 ANSI 之类的编码。

由于 Python 胶水语言的特性，大家在以后很有可能遇到用 Python 处理文件的需求，这时候字符编码将会是绕不开的话题，这里建议大家亲自上手尝试一下。

下面是一些编码解码的例子：

```
In [1]: 'Hello world'.encode()
Out[1]: b'Hello world'

In [2]: b'Hello world'.decode()
Out[2]: 'Hello world'

In [3]: '你好'.encode()
Out[3]: b'\xe4\xbd\xa0\xe5\xa5\xbd'

In [4]: '你好'.encode('utf-8')
Out[4]: b'\xe4\xbd\xa0\xe5\xa5\xbd'

In [5]: '你好'.encode('gbk')
Out[5]: b'\xc4\xe3\xba\xc3'

In [6]: b'\xe4\xbd\xa0\xe5\xa5\xbd'.decode('utf-8')    # 使用了正确的格式解码
Out[6]: '你好'

In [7]: b'\xe4\xbd\xa0\xe5\xa5\xbd'.decode('gbk')    # 使用了错误的格式解码
Out[7]: '浣犲ソ'
```

4 Python 容器

现在我们进入容器的章节。容器可以类比 C++ 的 STL，提供了一些良好封装的类。这里有一些预备知识：

- `type(obj)`：获取 obj 的类型
- `isinstance(obj, class_or_tuple)`：判断 obj 是否是对应的 class 的实例
- `id(obj)`：获取变量的唯一编号，可以理解为获取它的地址。

4.1 `list`

`list` 列表可以大致理解为 C++ 中的数组或 Vector。但是它对元素类型没有任何要求。

```
In [1]: a = [1, 'hi', 3.0]

In [2]: a
Out[2]: [1, 'hi', 3.0]

In [3]: a[2]
Out[3]: 3.0

In [4]: id(a)
Out[4]: 4411219264

In [5]: a.append(999)

In [6]: id(a)
Out[6]: 4411219264

In [7]: a
Out[7]: [1, 'hi', 3.0, 999]
```

注意到对 `a` 进行修改后它的 `id` 没有发生变化。对 a 进行赋值后会不会有变化呢？答案是会有的。从这里大家可以发现修改和赋值是有一定差异的。

列表还有一些内置的函数，例如 `count`，`sort`，`pop`，`remove`，这些大家可以在课后自行了解。

我们该怎么构造一个空的列表呢？下面两种方法都可以：

```
a = []
b = list()
```

在接下来要构造其他类型的空元素的时候，我们也可以采用类似的方法。

对列表的索引和 C++ 中基本一致，用方括号，序号从 0 开始。此外，`list` 还支持反向索引和切片的操作，例如

```
In [1]: a = [9, 0 , 3, 4]
```

```
   ...: a[-1]
Out[1]: 4

In [2]: a[-2]
Out[2]: 3

In [3]: a[1:3]
Out[3]: [0, 3]

In [4]: a[1:3:-1]
Out[4]: []

In [7]: a[3:1:-1]
Out[7]: [4, 3]

In [8]: a[::2]
Out[8]: [9, 3]
```

4.2 tuple

Python 的元组与列表类似，不同之处在于元组的元素不能修改。元组使用小括号，列表使用方括号。

元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

```
tup1 = () # 空元组
tup1 = (50, ) # 元组中只包含一个元素时，需要在元素后面添加逗号
tup1[0] = 20 # Error! Elements of tuple cannot be changed.
```

元组的小括号有时候可以省略的，比如

```
In [1]: a, b = 3, 4

In [2]: a
Out[2]: 3

In [3]: b, a = a, b

In [4]: a
Out[4]: 4
```

交换元素、返回多个值的时候元组会显得格外方便。

4.3 set

`set` 就是集合，其中的元素是无序的。可以用 `s.add()`，`s.remove()` 和 `in` 运算符完成其主要功能。构造的时候，可以用 `{}`，但是构造空集合只能用 `set()`，因为 `{}` 被字典所占用。

```
In [1]: s = {3, 5, 7}

In [2]: 3 in s, 4 in s
Out[2]: (True, False)

In [3]: s.add(8)

In [4]: s.remove(3)

In [5]: s
Out[5]: {5, 7, 8}
```

4.4 `dict`

字典是一个键值对容器，可以存储很多 `key->value` 的映射，其中 `value` 可以是任意变量类型，因此也可以嵌套；不过，对于 `key`，他必须是可 hash 的，因此可变容器不满足其要求。

```
In [1]: s = {}

In [2]: s["你好"] = "你很好"

In [3]: s
Out[3]: {'你好': '你很好'}

In [4]: s.get('唔')

In [5]: # 上面返回了一个 None

In [6]: s[2] = 'hi'

In [7]: s[{1,2}] = 111
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 s[{1,2}] = 111

TypeError: unhashable type: 'set'

In [8]: s["你好"]
Out[8]: '你很好'
```

注意到 dict 和 list 的嵌套可以很方便地转化为 json，这在实际处理环节中经常用到。

5 函数

Python 中函数的定义十分简单：

```
def myFunc(params):
    # Your Code...
    return
```

和控制语句一样，也是通过缩进来区分代码块；如果没有显示的返回语句，或者没有返回值，则默认返回 None；由于元组的存在，我们可以同时返回多个值。

这里有一个有些滑稽，却又真实存在的问题：我希望定义一个空的函数怎么办呢？答案是这样：

```
def doNothing():
    pass
```

我们可以在函数定义行末尾添加上它的返回值，但这不会对代码运行造成任何影响与限制，只是给程序员自己看的而已。

```
def needString() -> str:
    ret = 3
    return ret    # No Error
```

5.1 函数参数

Python 对于函数参数的支持是很灵活的。Python 中支持四类参数：位置参数，关键字参数，默认参数和可变参数。

5.1.1 位置参数

位置参数是指函数调用时按照函数定义时的参数顺序传递的参数

```
def greet(name, message):
    print(f"{message}, {name}!")

greet("Alice", "Hello")  # 输出 "Hello, Alice!"
```

5.1.2 关键字参数

关键字参数是指在函数调用时使用参数名来指定参数的值。这样可以不按照函数定义时的参数顺序传递参数，而是根据参数名来确定参数值。

```
greet(message="Hi", name="Bob")  # 输出 "Hi, Bob!"
```

前两者可以混用，位置参数一定要在关键字参数的前面。

5.1.3 默认参数

这个还比较好理解，就是给一些参数默认值

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")

greet("Alice")  # 输出 "Hello, Alice!"
greet("Bob", "Hi")  # 输出 "Hi, Bob!"
```

5.1.4 可变参数

在Python中，可变参数有两种：`*args` 和 `**kwargs`。我们先来看前者

`*args` 表示任意多个位置参数，它是一个元组类型：

```
def add(*args):
    result = 0
    for num in args:
        result += num
    return result

print(add(1, 2, 3))  # 输出 6
print(add(1, 2, 3, 4, 5))  # 输出 15
```

再来看一下后者 `**kwargs`，它表示多个关键字参数，实际上是一个字典类型，不过在传入的时候我们也不需要显示写成字典。

```
def greet(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

greet(name="Alice", message="Hello")  # 输出 "name: Alice" 和 "message: Hello"
greet(name="Bob", age=30, city="New York")  # 输出 "name: Bob"、"age: 30" 和 "city: New York"
```

这些不同的参数类型可否混用呢？请大家自行实验

6 面向对象

Python中的类基础用法比C/C++更简洁一些，这里给大家一些例子，希望大家能通过这些例子来学习

```python
class myClass:
    pass  # 定义了一个空类

myClass.b = 1   # 为 myClass 添加类公有成员变量 b

c = myClass()   # 实例化
print(c.b)      # Output: 1

c.b = 3

print(c.b)      # Output: 3
print(myClass.b) # Output: 1
print(c.__class__.b)  # Output: 1
```

这边有些神奇。我们先为 myClass 定义了类变量；当我们将类实例化的时候，会默认共享这一个变量，指向同一个地址；但是，如果在某个对象上修改了类变量的值，那么这个对象上的类变量会变成实例变量，即该对象会拥有自己的属性，而不再共享类变量；此时，我们仍然可以通过 `instance.__class__.var` 来访问共享变量。

出于 Python 动态语言的特性，我们可以随时为实例添加成员变量。

我们再来看一下函数

```python
class A:
    def __init__(self, a, b):
        self.a = a
        self.__b = b

    def print(self):
        print(self.a, self.__b)

a = A(1, 2)
print(a.a)    # Output: 1
print(a.__b)  # Output: AttributeError: 'A' object has no attribute '__b'
a.print()     # Output: 1 2
```

`__init__` 是构造函数，这个学过 C++ 的同学应该还比较熟悉，但是要注意 Python 中只能有一个构造函数，同时大部分情况下我们不会用到析构函数；`print` 就是我们自己定义的一个成员函数。

所有成员函数第一个参数必须是 self，它起到和 C++ 中 this 类似的作用。不加 self 的变量、函数会被视作局部或全局的变量、函数，但是不会视作类成员，你需要显示地指出 self 。

注意到，我们为这个类添加了两个成员变量 `a` 和 `__b`，大家从下面的输出中也可以看出来，以 `__` 开头的默认就是私有成员变量，其他的都是公有成员变量。

继承的基础语法是 `class A(parent_class)`，默认是公有继承。请大家自学。

7 生成式

生成式（Comprehension）是一种简洁的语法，用于快速创建列表、字典、集合等数据结构。它可以在一行代码中生成一个新的数据结构，非常方便。

```
In [1]: a = list(range(10))

In [2]: a
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: [i ** 3 for i in range(4)]
Out[3]: [0, 1, 8, 27]

In [4]: {i * 2 for i in range(10) if i % 8 != 0}
Out[4]: {2, 4, 6, 8, 10, 12, 14, 18}

In [5]: {str(i): i for i in range(10) if i % 8 != 0}
Out[5]: {'1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '9': 9}

# 可以尝试一下，该怎么构造元组呢?
```

总结一下

1. 生成式由这三部分组成：元素 + for 循环 + [筛选条件]
2. 可以生成集合、字典、列表、元素，用相应的符号即可

现在 Python 基础语法已经学完了，大家只要多写写就应该能掌握了。不过，Python 真正的魅力在于它庞大的生态；要进入庞大的生态，有一些进阶工具需要学习与配置。

8 包管理

8.1 import

Python 的一大优势在于它的第三方库真的很多，使得很多事情很容易完成。例如说，你想要做一个填表的网站，完成后将用户的信息填入 Word 文档内发还给客户，这一过程如果自己编写代码会是一件很复杂的事情，而 Python 就有完善的第三方库帮助我们完成这件事情。

无论是 Python 自带的库还是第三方库，引入方法均如下；

```
import myModule;  # 引入整个库
# 后续调用使用 myModule.funcName()
import myModule as mM;  # 起别名

from myModule import some;  # 引入部分
# 后续调用使用 some()
from myModule import this, that;  # 同时引入多个部分
from myModule import *;     # 引入全部，不推荐
```

在使用一个叫做 `tqdm` 的包的时候，往往需要 `from tqdm import tqdm` ，大家在后续实践中可以深刻体会到为什么要这么干（笑）。

8.2 pip

Python 自带了包管理器 pip, 安装第三方库十分简单。一般情况下，只用输入 `pip install <包名>` 就可以安装第三方包了（也可能需要使用 `pip3`）。

如果你因为中国大陆的网络原因无法顺畅地安装第三方包，可以根据 这个链接 进行换源。

8.3 标准库

Python内置了很多标准库，这些库提供了各种各样的功能，可以帮助我们快速开发Python应用程序。下面是Python中一些常用的标准库。时间原因，我们无法在这里详细介绍这些库，仅做列举：

1. os：提供了与操作系统交互的功能，如文件操作、目录操作、进程管理等。
2. sys：提供了一些与Python解释器和Python运行环境相关的功能，如命令行参数、标准输入输出、环境变量等。
3. datetime：提供了处理日期和时间的功能，如日期的计算、格式化、解析等。
4. re：提供了正则表达式的支持，可以用来进行文本匹配和替换等操作。
5. math：提供了数学运算的函数库，如三角函数、指数函数、常量等。
6. random：提供了生成随机数的函数，可以用来进行模拟和随机抽样等操作。
7. json：提供了JSON格式的编码和解码功能，可以进行数据的序列化和反序列化。
8. sqlite3：提供了SQLite数据库的支持，可以进行数据库的连接、查询、修改等操作。

这些标准库都是Python自带的，无需额外安装，可以直接在Python中使用。在编写Python应用程序时，使用这些标准库可以大大提高我们的开发效率，避免重复造轮子。

8.4 常用第三方库

当今，Python 生态系统中有数以千计的第三方库，其中许多库都非常有用。以下是一些常用的 Python 第三方库：

1. NumPy：用于数学计算和科学计算的库，效率较高。
2. Pandas：提供了数据结构和数据分析工具，特别是在数据处理方面非常有用。
3. Matplotlib：用于绘制静态、动态和交互式图形的绘图库。
4. TensorFlow：用于机器学习和深度学习的库。
5. PyTorch：另一个流行的深度学习框架。
6. Django：用于编写 Web 应用程序的全功能框架，提供了许多开箱即用的功能和工具。

7. Requests：用于发送 HTTP 请求和处理响应的库，非常适合编写 Web 应用程序和爬虫。

这仅仅是 Python 生态系统中的一小部分。还有更多的库等待大家的探寻。

8.5 我该怎么学习使用一个库

1. 阅读文档
2. 尝试样例、经典项目
3. 根据 IDE / 编辑器的提示阅读接口文档
4. 问 ChatGPT

9 环境配置进阶

9.1 Conda

我们刚刚介绍了 Python 有很多现成的包，这些包被广泛应用。事实上，任何大型的 Python 项目或多或少都引入了第三方的包。然而，这些包的不同版本之间有可能是不互相兼容的。如果我们同时开发 A 项目和 B 项目，A 项目要求 x 包的 4.0 版本， B 项目却只能在 x 包的 5.0 版本下运行，这就会给我们带来极大的困扰。

Conda 的一大作用就是创建独立的虚拟环境，每个环境都可以拥有自己的 Python 解释器、库和依赖项。这使得在不同的项目或应用程序之间隔离依赖项变得容易。我们可以为每一个项目创建专属的虚拟环境，在切换项目的同时也切换虚拟环境，避免依赖上的冲突。

Conda 的安装相对简单，请大家自行去 Tuna 下载系统对应版本的 miniconda 并安装（请安装较新版本的 Conda）。下面有一些笔者常用的操作供参考：

1. `conda create -n <env_name> python=3.9` 创建一个 Python 3.9 版本的新虚拟环境
2. `conda activate <env_name>` 激活这一虚拟环境。激活后，你可以正常使用 `pip` 安装包，所安装的包会且只会作用于当前的虚拟环境
3. `pip freeze > requirements.txt` 将当前环境中 pip 安装的包的列表存储到 requirements.txt 当中
4. `pip install -r requirements.txt` 按照 requirements.txt 中的要求在当前环境中安装对应的包
5. `conda deactivate` 退出当前的虚拟环境

当一个项目需要多人协作的时候，流程往往是这样的：

- 一个人：创建新的虚拟环境并激活 -> 安装必要的包 -> 导出到 requirements.txt 中，一并上传至项目仓库
- 其他人：把仓库拉下来 -> 创建新的虚拟环境并激活 -> 根据 requirements.txt 安装需要的包

事实上 conda 自身也可以进行包管理，更为正确的做法也是使用 `environment.yml`，~~不过由于讲师自己软王也是摆烂直接 pip 的所以就不讲了。~~感兴趣的同学可以自己查询。

9.2 IPython, IDE

IPython（Interactive Python）是一个增强版的 Python 解释器，它提供了许多增强的交互式功能，如代码自动补全、历史记录、对象内省、交互式数据可视化等。事实上，细心的同学应该发现了，我在课上使用的更多就是 IPython 而非 Python 自身的解释器。大家可以参考赵晨阳学长 2022 年的暑培讲义去了解更多。**你也可以在这一份讲义中了解到什么是 Pythonic, 什么是良好的 Python 编程风格。**

IDE 往往也能实现 IPython 的大部分功能，让你的开发效率更上一层楼。大家常用的开发环境大概是VSCode + 插件，这里推荐一下 PyCharm, 是一款很不错的 IDE，清华的同学也可以免费取得学生正版。

9.3 Jupyter Notebook

Jupyter Notebook（前身为 IPython Notebook）是一个基于 Web 的交互式计算环境，可以在其中创建、编辑和共享文档，包括代码、文本、图形、可视化和数学公式等。大家可以将它看作一个升级版本的 IPython notebook。ChatGPT 告诉我，它主要有以下优势：

1. Web 界面：Jupyter Notebook 提供了一个基于 Web 的用户界面，可以在其中创建、编辑和运行代码、文本和图形等。这使得 Jupyter Notebook 可以在多种设备和平台上运行，包括台式机、笔记本电脑和移动设备。
2. 交互性：Jupyter Notebook 支持交互式计算，可以快速测试和调试代码，并在文档中直接显示结果。这使得数据分析和可视化变得更加容易。
3. 可视化支持：Jupyter Notebook 支持多种可视化库，包括 Matplotlib、Bokeh、Plotly 等。这使得用户可以轻松地创建交互式图表和可视化结果。
4. 共享性：Jupyter Notebook 支持导出为多种格式，包括 HTML、PDF、Markdown 等。这使得用户可以轻松地共享文档和分析结果。

总的来说，Jupyter Notebook 是一个非常强大、灵活和易于使用的交互式计算环境，适用于数据分析、机器学习、科学计算、文档编写和教学等多个场景。

10 作业

「文章填词」是一个十分简单的小游戏。具体来说，出题者事先准备好一篇文章，并将其中的一些单词挖去；对于挖去的单词，出题者会给一定提示，例如该词的词性、褒贬、类型等；做题者看不到文章，只能根据提示随意选择单词；最终将做题者给定的单词填回原文，往往会达成不错的喜剧效果。你可以和他人比拼，谁能填出最正常/无厘头的文章，同时，这也能作为学习外语、拓宽词汇量的途径。下面是一个小例子：

> 中文题目：
>
> 我们都爱 {{1}} 这门课程。这门课程是多么的 {{2}}，以至于所有人都在课程上认真地 {{3}}。在设计数字电路时,我们需要运用到逻辑门、半导体存储器等知识。这些内容相互联系,共同构成复杂的数字电路。这门课能启发我们的逻辑思维能力和科学思考能力。通过为难我们的练习和作业,我们的理解能力和解决问题的能力得到了提高。这些将对今后的 {{4}} 和工作有很大益处。
>
> 单词限制：

1. 教材名称　　2. 形容词　　3. 动词，与学生相关　　4. 你最喜欢做的事情

填空可能是：

1. 《数字逻辑电路》　　2. 青涩　　3. 内卷　　4. 玩原神

10.1 基础功能

程序主体可以大致分为以下几个步骤：

- 启动：根据命令行参数启动，命令行参数内至少包含指定题库的文件路径、是否指定文章（如果指定，那么指定哪一篇）两个参数。【需要自学 `argparse` 这一库】
- 读取：读取相应的题库，解析 JSON 文件 【需要自学 `json` 这一库】，选择一份文章。
- 输入
- 替换：将用户的输入填入文章当中【可能需要用到 `re` 这一库】
- 显示：可以直接在命令行打印，也可以用其他更加优美的方式展现。

我们在 仓库 里提供了一个半成品代码，完成了整体的架构并进行了部分的实现，供同学参考学习。注意，我们提供了 `requirements.txt` ，不过它目前是空的，若你使用了其他第三方的库，也请更新该文件。当然，你完全可以从零开始，自己编写代码。

10.2 拓展功能

基础功能的要求较为简单，且没有涉及到第三方库。推荐同学们完成一些拓展功能，更深入感受 Python 强大的生态与开发效率。对于以下功能，请自行寻找合适的解决方案、第三方库，务必不要尝试自己从零开始解决。

- GUI. 推荐自学 `Streamlit` 这一第三方库，可以极其迅速地构建轻量级而优美的图形界面。你可以在图形界面里添加选择文章等等实用的功能 **【推荐】**。
- 鲁棒性. 当参数、用户输入或题库文件不合法的时候，直接报错退出的做法不太优雅。你可以尝试学习 Python 中 `try... except` 的用法，达成更好的错误处理。
- ~~更多题库. 总不至于大家都用我给的这边傻傻的文章当题目吧。~~
- 语言检测. 可以添加命令行参数，限制输入单词的语言，当输入语言与要求不一致的时候拒绝输入。
- 首字母限制. 对于英文，我们可以限制单词的首字母；对于中文，可以尝试限制单词首字的起始音素。
- 保存. 用户可以将自己得到的文章通过任意方式保存起来。
- 出题功能. 用户无需自行编辑 JSON 文件，而是可以在程序提供的窗口内出题。
- 更多有趣的功能！

10.3 作业提交

请在 原仓库 里通过提 Issue 的方式提交你的作业仓库链接，记得附上你的个人信息。

并且请在你的作业仓库里以 README.md 的形式附上一份较为完整的使用说明，包括但不限于

- 项目环境 / 如何启动
- 使用配置（启动参数、文章文件格式等）
- 完成功能 / 游戏说明

Deadline 为一周后 (23.07.25 23:59)

---

11 附录

以下部分摘自过去两年的讲义，给了一些常用(库)的使用方法：

11.1 文件操作

首先相信大家都有了 C 文件操作基础(以及之前讲 Linux 应该也有说)，因此这里略过相对路径、打开方式等不谈，首先在终端中执行指令构造一个简单文件:

`echo -e "123\n456\n789" > temp.txt（或者手动创建，第一行123，第二行456，第三行789）`

然后再到Python中进行操作:

```
file = open('temp.txt')
line1 = file.readline()
lines = file.readlines()
file.close()
print(line1)
print(lines)
```

这里上下文管理器(也就是 `with`)，在读写文件时就非常有用，可以避免由于忘记关闭文件导致丢失数据:

```
with open('temp.txt') as file:
    line1 = file.readline()
    lines = file.readlines()
# 这里 file 会被自动 close 掉

print(line1)
print(lines)
```

不过这还是太简单了点，下面用 bash 加点料: `echo -e "你好世界" >> temp.txt`

再到 Python 操作:

```
data = open('temp.txt').read() # 此时有没有显式关闭并不重要，因为临时变量被析构时文件数据也被清除了
print(data)
```

它执行成功了吗?我不知道，因为我无法确定你的终端编码与你的Python默认编码是否一致(事实上在 Windows 的终端中执行 echo -e 是错误的)。在我的电脑上它可能等价于:

```
data = open('temp.txt', encoding='utf-8').read()
print(data)
```

但是在 Windows 电脑上，它或许等价于

```
data = open('temp.txt', encoding='gbk').read()
print(data)
```

Anyway，你应当自己保证或让提供文件的人保证文件的编码是什么，否则你将需要一些手段(库)来推测编码。

此外，介绍一点比较高效的操作:

```
with open('temp.txt') as file:
    for line in file: # 使用迭代器按行读取，效率比全文读取高得多，想象一下如果这文件几个G
        print(line)
```

11.2 JSON

考虑一下如何把一个字典保存在文件里?

```
d = {'姓名': '张三', '年龄': 18, '生日': None}
```

这就是一个字典，直接写到文件里是行不通的，文件只能写字节串或者字符串，你需要转化它，一个可能的方法是用 `str(d)` ，如果你想到了这个操作，那很好，但是还不够，因为我们缺少一个手段将字符串转化回字典(这被称作序列化和反序列化)。json库正为我们提供了这样的手段:

```
# 引入整个库
import json
json.dump(d, open('data.json', 'w')) data = json.load(open('data.json'))
```

或许你会很好奇data.json里面是些什么:

```
{"\u59d3\u540d": "\u5f20\u4e09", "\u5e74\u9f84": 18, "\u751f\u65e5": null}
```

哦这是 Unicode，真不是给人看的东西.....不过没关系，我们很容易把它改成给人看的:

```
json.dump(d, open('data.json', 'w'), ensure_ascii=False)
```

再来看看文件里的东西吧:

```
{"姓名": "张三", "年龄": 18, "生日": null}
```

这次看上去是不是好多了?注意那个 `null` ，这是 JavaScript的写法，毕竟 json 的全称是 JavaScript Object Notation。

与 dump 相反的操作是 load，也请大家查阅相关资料。

11.3 tqdm

很方便地帮助我们监视程序执行进度，应用也十分简单：

```
In [1]: from tqdm import tqdm;

In [2]: for i in tqdm(range(10000000)):
   ...:     pass
   ...:
100%|██████████████████████████████████| 10000000/10000000 [00:01<00:00, 7445855.48it/s]
```

更多的库还请大家在实践中探索。