[1 python编程基础 4](#_Toc204528035)

[1.1 Python基础语法 4](#_Toc204528036)

[1.1.1 数据类型 4](#_Toc204528037)

[1.1.1.1 数字 4](#_Toc204528038)

[1.1.1.2 字符串 5](#_Toc204528039)

[1.1.1.3 布尔（bool） 5](#_Toc204528040)

[1.1.1.4 列表List 6](#_Toc204528041)

[1.1.1.5 元组 7](#_Toc204528042)

[1.1.1.6 集合Set 7](#_Toc204528043)

[1.1.1.7 字典Dictionary 7](#_Toc204528044)

[1.1.1.8 bytes 7](#_Toc204528045)

[1.1.1.9 数据类型转换 8](#_Toc204528046)

[1.1.2 变量与常量 8](#_Toc204528047)

[1.1.2.1 变量 8](#_Toc204528048)

[1.1.2.2 常量 9](#_Toc204528049)

[1.1.3 运算符 9](#_Toc204528050)

[1.1.3.1 算术运算符 9](#_Toc204528051)

[1.1.3.2 比较运算符 10](#_Toc204528052)

[1.1.3.3 赋值运算符 10](#_Toc204528053)

[1.1.3.4 位运算符 10](#_Toc204528054)

[1.1.3.5 逻辑运算符 11](#_Toc204528055)

[1.1.3.6 成员运算符 11](#_Toc204528056)

[1.1.3.7 身份运算符 11](#_Toc204528057)

[1.1.3.8 运算符优先级 11](#_Toc204528058)

[1.1.4 输入输出 12](#_Toc204528059)

[1.1.4.1 输入 12](#_Toc204528060)

[1.1.4.2 输出 12](#_Toc204528061)

[1.1.4.3 输出格式化 13](#_Toc204528062)

[1.1.5 条件语句 13](#_Toc204528063)

[1.1.5.1 if语句 13](#_Toc204528064)

[1.1.5.2 match..case 14](#_Toc204528065)

[1.1.6 循环语句 14](#_Toc204528066)

[1.1.6.1 while循环 14](#_Toc204528067)

[1.1.6.2 for循环 14](#_Toc204528068)

[1.1.6.3 break与continue 15](#_Toc204528069)

[1.1.7 命名空间与作用域 15](#_Toc204528070)

[1.1.7.1 命名空间 15](#_Toc204528071)

[1.1.7.2 作用域 16](#_Toc204528072)

[1.1.7.3 全局变量和局部变量 17](#_Toc204528073)

[1.2 Python数据结构 18](#_Toc204528074)

[1.2.1 字符串 18](#_Toc204528075)

[1.2.1.1 访问字符串 18](#_Toc204528076)

[1.2.1.2 字符串转义 18](#_Toc204528077)

[1.2.1.3 字符串运算 19](#_Toc204528078)

[1.2.1.4 格式字符串 19](#_Toc204528079)

[1.2.1.5 三引号 19](#_Toc204528080)

[1.2.1.6 f-string 19](#_Toc204528081)

[1.2.2 列表 20](#_Toc204528082)

[1.2.2.1 访问列表 20](#_Toc204528083)

[1.2.2.2 更新列表 20](#_Toc204528084)

[1.2.2.3 删除列表 21](#_Toc204528085)

[1.2.2.4 列表操作符 21](#_Toc204528086)

[1.2.2.5 列表截取与拼接 21](#_Toc204528087)

[1.2.2.6 嵌套列表 21](#_Toc204528088)

[1.2.2.7 列表函数方法 21](#_Toc204528089)

[1.2.3 元组 22](#_Toc204528090)

[1.2.3.1 访问元组 22](#_Toc204528091)

[1.2.3.2 修改/删除元组 22](#_Toc204528092)

[1.2.3.3 元组运算 23](#_Toc204528093)

[1.2.3.4 元组索引与截取 23](#_Toc204528094)

[1.2.3.5 元组内置函数 23](#_Toc204528095)

[1.2.4 字典 23](#_Toc204528096)

[1.2.4.1 创建空字典 23](#_Toc204528097)

[1.2.4.2 访问字典 23](#_Toc204528098)

[1.2.4.3 修改字典 23](#_Toc204528099)

[1.2.4.4 删除字典元素 24](#_Toc204528100)

[1.2.4.5 字典键的特性 24](#_Toc204528101)

[1.2.4.6 字典内置函数方法 24](#_Toc204528102)

[1.2.5 集合 25](#_Toc204528103)

[1.2.6 推导式 25](#_Toc204528104)

[1.2.7 列表推导式 26](#_Toc204528105)

[1.2.8 字典推导式 26](#_Toc204528106)

[1.2.9 集合推导式 26](#_Toc204528107)

[1.2.10 元组推导式（生成器表达式） 26](#_Toc204528108)

[1.2.11 其它数据结构 27](#_Toc204528109)

[1.2.11.1 栈 27](#_Toc204528110)

[1.2.11.2 队列 28](#_Toc204528111)

[1.3 Python函数与模块 29](#_Toc204528112)

[1.3.1 函数 29](#_Toc204528113)

[1.3.1.1 定义函数 29](#_Toc204528114)

[1.3.1.2 函数调用 29](#_Toc204528115)

[1.3.1.3 参数传递之对象传递方式 29](#_Toc204528116)

[1.3.1.4 参数传递之参数类型 30](#_Toc204528117)

[1.3.1.5 匿名函数lambda 31](#_Toc204528118)

[1.3.1.6 其它特殊函数 31](#_Toc204528119)

[1.3.2 模块 31](#_Toc204528120)

[1.3.2.1 模块导入 32](#_Toc204528121)

[1.3.2.2 \_\_name\_\_ 属性 32](#_Toc204528122)

[1.3.2.3 标准模块 33](#_Toc204528123)

[1.3.2.4 包 33](#_Toc204528124)

[1.4 Python面向对象OOP 33](#_Toc204528125)

[1.4.1 面向对象简介 33](#_Toc204528126)

[1.4.2 类和对象 34](#_Toc204528127)

[1.4.2.1 类 34](#_Toc204528128)

[1.4.2.2 对象 34](#_Toc204528129)

[1.4.3 类方法 35](#_Toc204528130)

[1.4.3.1 构造方法\_\_init\_\_ 35](#_Toc204528131)

[1.4.3.2 静态方法与类方法 35](#_Toc204528132)

[1.4.4 类中的属性 35](#_Toc204528133)

[1.4.5 继承 36](#_Toc204528134)

[1.4.5.1 多继承 36](#_Toc204528135)

[1.4.5.2 重写与多态 37](#_Toc204528136)

[1.4.6 封装 37](#_Toc204528137)

[1.4.7 抽象 38](#_Toc204528138)

[1.5 Python错误与异常处理 38](#_Toc204528139)

[1.5.1 区分错误与异常 38](#_Toc204528140)

[1.5.2 异常处理 38](#_Toc204528141)

[1.5.3 常见异常 39](#_Toc204528142)

[1.5.4 自定义异常 39](#_Toc204528143)

[1.5.5 断言assert 40](#_Toc204528144)

[1.6 Python文件操作 40](#_Toc204528145)

[1.6.1 打开文件 40](#_Toc204528146)

[1.6.2 读写文件 42](#_Toc204528147)

[1.6.3 file对象 43](#_Toc204528148)

[1.7 Python进阶 43](#_Toc204528149)

[1.7.1 Python迭代器与生成器 43](#_Toc204528150)

[1.7.2 Python装饰器 45](#_Toc204528151)

[1.7.2.1 装饰器基础 45](#_Toc204528152)

[1.7.2.2 带参数的装饰器（工厂） 46](#_Toc204528153)

[1.7.2.3 装饰器的嵌套 46](#_Toc204528154)

[1.7.3 上下文管理器 47](#_Toc204528155)

[1.7.3.1 上下文管理器的实现 47](#_Toc204528156)

[1.7.3.2 with语句 47](#_Toc204528157)

[1.7.3.3 以文件操作为例子 47](#_Toc204528158)

[1.7.4 多线程 48](#_Toc204528159)

[1.8 常用库与框架 48](#_Toc204528160)

[1.8.1 PyMySQL 48](#_Toc204528161)

[1.8.2 Json 48](#_Toc204528162)

[1.8.3 Requests 48](#_Toc204528163)

[1.8.3.1 核心原理 48](#_Toc204528164)

[1.8.3.2 请求方法 48](#_Toc204528165)

[1.8.3.3 请求参数 49](#_Toc204528166)

[1.8.3.4 响应内容 54](#_Toc204528167)

[1.8.3.5 Session与持久化配置 56](#_Toc204528168)

[1.8.4 Selenium/Playwright 57](#_Toc204528169)

[1.8.4.1 Selenium 57](#_Toc204528170)

[1.8.4.2 Playwright 61](#_Toc204528171)

[1.8.5 pytest 64](#_Toc204528172)

# python编程基础

Python是一种高级、通用的​​解释型编程语言​​，设计理念强调代码简洁性、可读性与开发效率。

## Python基础语法

### 数据类型

**标准数据类型：**

Python3 中常见的数据类型有：

* Number（数字）
* String（字符串）
* bool（布尔类型）
* List（列表）
* Tuple（元组）
* Set（集合）
* Dictionary（字典）

**Python3 的六个标准数据类型中：**

* **不可变数据**（3 个）：**Number**（数字）、**String**（字符串）、**Tuple**（元组）；
* **可变数据**（3 个）：**List**（列表）、**Dictionary**（字典）、**Set**（集合）。

此外还有一些高级的数据类型，如: 字节数组类型(bytes)。

#### 数字

Python3 支持 int、float、bool、complex（复数），只有一种整数类型 int，表示为长整型。

内置type() 函数可以用来查询变量所指的对象类型，此外还可以用 isinstance 来判断。

isinstance() 和 type() 的区别在于：

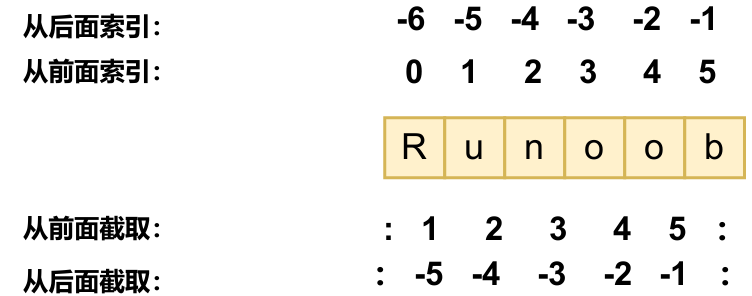
* type()不会认为子类是一种父类类型。
* isinstance()会认为子类是一种父类类型。

**注意：Python3 中，bool 是 int 的子类，True 和 False 可以和数字相加， True==1、False==0 会返回 True，但可以通过 is 来判断类型。**

#### 字符串

Python中的字符串用单引号 **'** 或双引号 **"** 括起来，同时使用反斜杠 **\** 转义特殊字符。

* 字符串的截取的语法格式如：变量[头下标:尾下标]



* 加号 + 是字符串的连接符；星号 \* 表示复制当前字符串，与之结合的数字为复制的次数。
* Python 使用反斜杠 \ 转义特殊字符，如果你不想让反斜杠发生转义，可以在字符串前面添加一个 r，表示原始 字符串。另外，反斜杠(\)可以作为续行符，表示下一行是上一行的延续。也可以使用 """...""" 或者 '''...''' 跨越多行

注意：

* 反斜杠可以用来转义，使用r可以让反斜杠不发生转义。
* 字符串可以用**+运算符连接**在一起，用**\*运算符重复**。
* Python中的字符串有两种索引方式，**从左往右以0开始**，**从右往左以-1开始**。
* Python中的字符串不能改变。

#### 布尔（bool）

布尔类型即 **True** 或 **False**。 在 Python 中，True 和 False 都是关键字，表示布尔值。

布尔类型特点：

* 布尔类型只有两个值：True 和 False。
* **bool 是 int 的子类**，因此布尔值可以被看作整数来使用，其中 True 等价于 1。
* 布尔类型可以和其他数据类型进行比较，比如数字、字符串等。在比较时，Python 会**将 True 视为 1，False视为 0**。
* 布尔类型可以和逻辑运算符一起使用，包括 and、or 和 not。这些运算符可以用来组合多个布尔表达式，生成一个新的布尔值。
* 布尔类型也可以被转换成其他数据类型，比如整数、浮点数和字符串。在转换时，True 会被转换成 1，Fals e 会被转换成 0。
* 可以使用 bool() 函数将其他类型的值转换为布尔值。以下值在转换为布尔值时为 False：None、False、零 (0、0.0、0j)、空序列（如 ''、()、[]）和空映射（如 {}）。其他所有值转换为布尔值时均为 True。

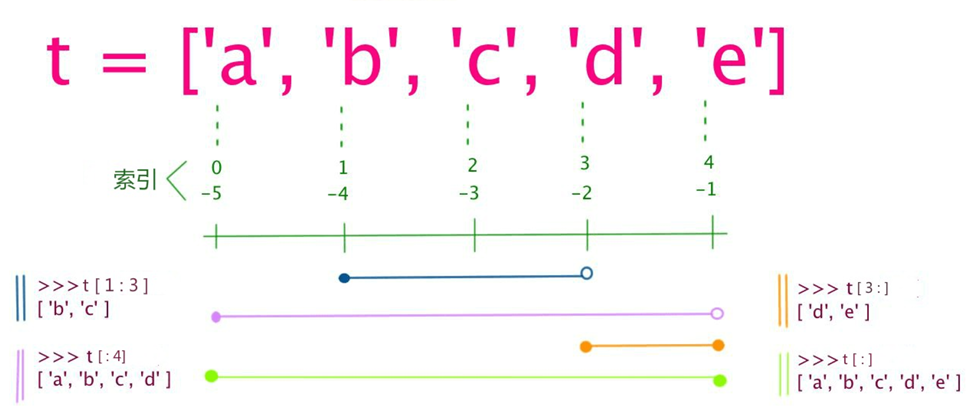
注意: 在 Python 中，所有非零的数字和非空的字符串、列表、元组等数据类型都被视为 True，只有 0、空字符串、空列表、空元组等被视为 False。因此，在进行布尔类型转换时，需要注意数据类型的真假性。

#### 列表List

列表是写在方括号 [] 之间、用逗号分隔开的元素列表。。列表中元素的类型可以不相同，它支持数字，字符串甚至可以包含 列表（所谓嵌套）。

列表截取的语法格式如：变量[头下标:尾下标]

索引值以 0 为开始值，-1 为从末尾的开始位置。

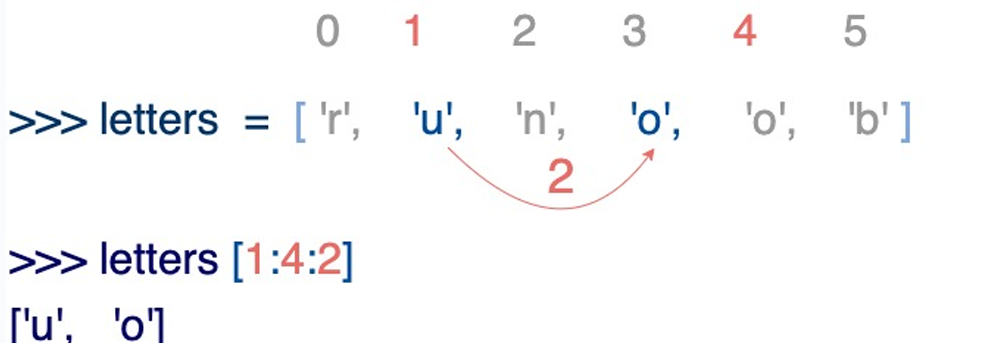


字符串可视为某种特殊的列表，因此其加号 + 是列表连接运算符，星号 \* 是重复操作。但是，与Python字符串不一样的是，列表中的元素是可以改变的，即可以通过索引修改列表内的内容。

注意：

1. 列表写在方括号之间，元素用逗号隔开。
2. 和字符串一样，列表可以被索引和切片。
3. 列表可以使用 + 操作符进行拼接。
4. 列表中的元素是可以改变的。

Python 列表截取可以接收第三个参数，参数作用是截取的步长。且如果第三个参数为负数表示逆向读取，可以用于翻转字符串。



#### 元组

元组（tuple）与列表类似，不同之处在于元组的元素不能修改。元组写在小括号 () 里，元素之间用逗号隔开，元组中的元素类型可以不相同。

 tuple = ( 'abcd', 786 , 2.23, 'runoob', 70.2  )

 tinytuple = (123, 'runoob')

元组与字符串类似，可以被索引且下标索引从0开始，-1 为从末尾开始的位置。也可以进行类似截取。其实可以把字符串看作一种特殊的元组。

注意：

1. 与字符串一样，元组的元素不能修改。
2. 元组也可以被索引和切片，方法一样。
3. 注意构造包含 0 或 1 个元素的元组的特殊语法规则。
4. 元组也可以使用 + 操作符进行拼接。

#### 集合Set

Python 中的集合（Set）是一种无序、可变的数据类型，用于存储唯一的元素。

sites = {'Google', 'Taobao', 'Runoob', 'Facebook', 'Zhihu', 'Baidu'}

* 集合中的元素不会重复，并且可以进行交集、并集、差集等常见的集合操作。
* 在 Python 中，集合使用大括号 {} 表示，元素之间用逗号 , 分隔。
* 另外，也可以使用 set() 函数创建集合。

注意：创建一个空集合必须用 set() 而不是 { }，因为 { } 是用来创建一个空字典。

#### 字典Dictionary

列表是有序的对象集合，字典是无序的对象集合。两者之间的区别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。

* 字典是一种映射类型，字典用 { } 标识，它是一个无序的 键(key) : 值(value) 的集合。

 tinydict = {'name': 'runoob','code':1, 'site': 'www.runoob.com'}

* 键(key)必须使用不可变类型。
* 在同一个字典中，键(key)必须是唯一的。

注意：

1. 字典是一种映射类型，它的元素是键值对。
2. 字典的关键字必须为不可变类型，且不能重复。
3. 创建空字典使用 { }。

#### bytes

bytes 类型表示的是不可变的二进制序列（byte sequence）。

* 与字符串类型不同的是，bytes 类型中的元素是整数值（0 到 255 之间的整数），而不是 Unicode 字符。bytes 类型通常用于处理二进制数据，比如图像文件、音频文件、视频文件等等。在网络编程中，也经常使用 bytes 类型来传输二进制数据。
* 与字符串类型类似，bytes 类型也支持许多操作和方法，如切片、拼接、查找、替换等等。同时，由于 bytes 类型 是不可变的，因此在进行修改操作时需要创建一个新的 bytes 对象。

#### 数据类型转换

* 隐式转化：自动完成

隐式类型转换中，Python 会自动将一种数据类型转换为另一种数据类型，不需要我们去干预。两种不同类型的数据进行运算，较低数据类型（整数）就会转换为较高数据类型（浮点数）以避免数据丢失。

* 显示转化：通过函数转化

|  |  |
| --- | --- |
| 函数 | 概述 |
| int(x[, base]) | 将x转换为一个整数（base为可选进制基数） |
| float(x) | 将x转换为一个浮点数 |
| complex(real[, imag]) | 用实部real和虚部imag创建一个复数 |
| str(x) | 将对象x转换为字符串 |
| repr(x) | 将对象x转换为表达式字符串 |
| eval(str) | 计算字符串中的有效Python表达式并返回对象 |
| tuple(s) | 将序列s转换为元组 |
| list(s) | 将序列s转换为列表 |
| set(s) | 转换为可变集合 |
| dict(d) | 创建字典，d需是(key,value)元组序列 |
| frozenset(s) | 转换为不可变集合 |
| chr(x) | 将整数x转换为对应ASCII字符 |
| ord(x) | 将字符x转换为对应的ASCII整数值 |
| hex(x) | 将整数x转换为十六进制字符串 |
| oct(x) | 将整数x转换为八进制字符串 |

### 变量与常量

#### 变量

变量是一个用来存储数据的标识符，可以在程序运行时发生变化。简单来说，变量就是一个给数据起的名字，它指向存储在内存中的数据对象。

* **动态类型**：在Python中，变量不需要预先声明数据类型，Python会根据赋值自动推导数据类型。Python会根据你赋给它的值自动识别变量类型，整数、浮点数、字符串、布尔值等都可以作为变量的值。

x = 10     # x 是一个整数

x = "Hello"  # x 变成了一个字符串

* **变量赋值**：Python中的变量赋值是通过 = 运算符来实现的。例如：

x = 5  # 将整数 5 赋值给变量 x

name = "Alice"  # 将字符串 "Alice" 赋值给变量 name

**命名规则**：

* 变量名只能包含字母、数字和下划线（\_），不能以数字开头。
* 变量名不能是Python的保留字（如 if, else, while, def 等）。
* 变量名区分大小写，x 和 X 是不同的变量。

**常用的命名风格**：

* 蛇形命名法（snake\_case）：用于变量和函数命名，单词间用下划线连接，如 my\_variable。
* 驼峰命名法（camelCase）：多用于类的实例变量和一些其他语言的习惯，如 myVariable。

#### 常量

常量是指在程序运行过程中其值不能改变的量。在Python中，虽然没有专门的“常量”类型，但常量通常是通过约定来实现的。

* **没有强制机制**：Python本身没有内建机制来强制使某个变量成为常量，所以一般通过约定来实现常量的行为。
* **命名约定：**通常在Python中，常量使用全大写字母来命名，以区分普通变量。

PI = 3.14159  # 常量 PI 存储圆周率

MAX\_VALUE = 1000  # 常量 MAX\_VALUE 存储最大值

* **不可修改的常量：**常量的值一旦赋值后，不应该被修改。在实际编程中，我们通过约定来保持常量的值不变，但Python本身不会阻止修改常量的值。

PI = 3.14159

PI = 3.14  # 在Python中，虽然这不符合常量的约定，但代码仍然可以执行

### 运算符

运算符是用于对变量或值执行特定操作的符号或关键字，它可以对操作数（即被运算的对象）进行数学计算、逻辑判断、位操作等处理。

Python 语言支持以下类型的运算符:

* 算术运算符
* 比较（关系）运算符
* 赋值运算符
* 逻辑运算符
* 位运算符
* 成员运算符
* 身份运算符
* 运算符优先级

#### 算术运算符

得到算数结果

|  |  |
| --- | --- |
| + | 加 - 两个对象相加 |
| - | 减 - 得到负数或是一个数减去另一个数 |
| \* | 乘 - 两个数相乘或是返回一个被重复若干次的字符串 |
| / | 除 - x 除以 y |
| % | 取模 - 返回除法的余数 |
| \*\* | 幂 - 返回x的y次幂 |
| // | 取整除 - 往小的方向取整数 |

#### 比较运算符

得到bool值

|  |  |
| --- | --- |
| == | 等于 - 比较对象是否相等 |
| != | 不等于 - 比较两个对象是否不相等 |
| > | 大于 - 返回x是否大于y |
| < | 小于 - 返回x是否小于y。 |
| >= | 大于等于 - 返回x是否大于等于y。 |
| <= | 小于等于 - 返回x是否小于等于y。 |

#### 赋值运算符

进行赋值，同时可以进行额外操作

|  |  |
| --- | --- |
| = | 简单的赋值运算符 |
| += | 加后赋值给自己 |
| -= | 减后赋值给自己 |
| /= | 除后赋值给自己 |
| %= | 求余后赋值给自己 |
| \*\*= | 求幂后赋值给自己 |
| //= | 整除后赋值给自己 |
| ：= | 先赋值后返回值 |

#### 位运算符

进行二进制层的操作

|  |  |  |
| --- | --- | --- |
| & | 按位与运算符：参与运算的两个值,如果两个相应位都为1, 则该位的结果为1,否则为0 | (a & b) 输出结果 12 ，二进制解释： 0000 1100 |
| | | 按位或运算符：只要对应的二个二进位有一个为1时，结 果位就为1。 | (a | b) 输出结果 61 ，二进制解释： 0011 1101 |
| ^ | 按位异或运算符：当两对应的二进位相异时，结果为1 | (a ^ b) 输出结果 49 ，二进制解释： 0011 0001 |
| ~ | 按位取反运算符：对数据的每个二进制位取反,即把1变为 0,把0变为1。~x 类似于 -x-1 | (~a ) 输出结果 -61 ，二进制解释： 1100 0011， 在一个有符号二进制数的补码形式。 |
| << | 左移动运算符：运算数的各二进位全部左移若干位，由" <<"右边的数指定移动的位数，高位丢弃，低位补0。左移以为等效于\*2，两位\*2\*2 | a << 2 输出结果 240 ，二进制解释： 1111 0000 |
| >> | 右移动运算符：把">>"左边的运算数的各二进位全部右移 若干位，">>"右边的数指定移动的位数，等效/2 | a >> 2 输出结果 15 ，二进制解释： 0000 1111 |

#### 逻辑运算符

即与或非，对bool进行运算

|  |  |  |
| --- | --- | --- |
| and | x and y | 布尔"与" - 如果 x 为 False，x and y 返回 x 的值，否则返回 y 的计 算值。 |
| or | x or y | 布尔"或" - 如果 x 是 True，它返回 x 的值，否则它返回 y 的计算 值。 |
| not | not x | 布尔"非" - 如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。 |

#### 成员运算符

判断是否包含成员

|  |  |
| --- | --- |
| In | 如果在指定的序列中找到值返回 True，否则返回 False。 |
| not in | 如果在指定的序列中没有找到值返回 True，否则返回 False。 |

#### 身份运算符

比较存储单元的对象

|  |  |  |
| --- | --- | --- |
| is | is 是判断两个标识符是不是引用 自一个对象 | x is y, 类似 id(x) == id(y) , 如果引用的是同一个对象则返回 True，否则返回 False |
| Is not | is not 是判断两个标识符是不是 引用自不同对象 | x is not y ， 类似 id(x) != id(y)。如果引用的不是同一个对象则返 回结果 True，否则返回 False。 |

注： id() 函数用于获取对象内存地址。

#### 运算符优先级

下表格列出了从最高到最低优先级的所有运算符， 相同单元格内的运算符具有相同优先级。 运算符均指二元运 算，除非特别指出。 相同单元格内的运算符从左至右分组（除了幂运算是从右至左分组）

|  |  |
| --- | --- |
| (expressions...), [expressions...], {key: value...}, {expressions...} | 圆括号的表达式 |
| x[index], x[index:index], x(arguments...), x.attribu te | 读取，切片，调用，属性引用 |
| await x | await 表达式 |
| \*\* | 乘方(指数) |
| +x, -x, ~x | 正，负，按位非 NOT |
| \*, @, /, //, % | 乘，矩阵乘，除，整除，取余 |
| +, - | 加和减 |
| <<,>> | 移位 |
| & | 按位与 AND |
| ^ | 按位异或 XOR |
| | | 按位或 OR |
| in,not in, is,is not, <=, >, >=, !=, == | 比较运算，包括成员检测和标识号检测 |
| not x | 逻辑非 NOT |
| and | 逻辑与 AND |
| or | 逻辑或 OR |
| if -- else | 条件表达式 |
| lambda | lambda 表达式 |
| := | 赋值表达式 |

### 输入输出

#### 输入

Python 中，输入方式的核心是 input() 函数。

input（””）函数输出内容并且等待内容输入，从标准输入读入一行文本，默认的标准输入是键盘，得到字符串。

 str = input("请输入：")

 print ("你输入的内容是: ", str)

#### 输出

Python两种输出值的方式: 表达式语句和 print() 函数。

* print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

参数列表：

1. \*objects：可以是多个要输出的对象（字符串、数字、变量等），可以同时传入多个对象，print() 会依次打印。
2. sep：指定多个对象之间的分隔符，默认为一个空格 ' '。
3. end：指定输出结尾的字符，默认为换行符 \n，即每次输出结束后都会换行。
4. file：指定输出的目标，默认输出到控制台，可以指定为文件等其他目标。
5. flush：指定是否立即刷新输出流，默认为 False，即缓冲输出。

Print()直接只输出字符串，可以通过str(),repr()来转化成字符串输出

str() 生成“用户友好的”字符串输出，而 repr() 用于生成“官方”的字符串输出，通常用于调试。

text = "Hello\nWorld"  # 含有换行符

print(str(text))  # 输出: Hello

                  #        World

print(repr(text))  # 输出: 'Hello\nWorld'  # 显示了换行符

#### 输出格式化

字符串拼接与格式化

name = "Alice"

age = 25

* + 拼接字符串（基本用法）：

print("My name is " + name + " and I am " + str(age) + " years old.")

* str.format()：

print("My name is {} and I am {} years old.".format(name, age))

* f-string（Python 3.6 及以上）：

print(f"My name is {name} and I am {age} years old.")

对齐和宽度

* str.format()的左、中、右、填充对齐：<、^、>、\*<

print("{:<10}".format("Hello"))  # 输出: 'Hello     '

print("{:>10}".format("Hello"))  # 输出: '     Hello'

print("{:^10}".format("Hello"))  # 输出: '  Hello   '

print("{:\*^10}".format("Hello"))  # 输出: '\*\*Hello\*\*\*'

* rjust(width, fillchar=' ')，ljust(width, fillchar=' ')，center(width, fillchar=' ')

rjust() 方法用于将字符串 右对齐，并将字符串左侧填充指定的字符，使其总长度为 width。如果字符串长度已经大于或等于 width，则不做任何改变。其它ljust即左对齐，center居中对齐。

text = "Hello"

print(text.rjust(10, "\*"))  # 输出: '\*\*\*\*\*Hello'

print(text.ljust(10, "\*"))  # 输出: 'Hello\*\*\*\*\*'

print(text.center(10, "\*"))  # 输出: '\*\*Hello\*\*\*'

### 条件语句

条件语句是通过一条或多条语句的执行结果（True 或者 False）来决定执行的代码块。

#### if语句

Python中if语句的一般形式如下所示：

if condition\_1:

    statement\_block\_1

elif condition\_2:

    statement\_block\_2

else:

    statement\_block\_

注意：

1. 每个条件后面要使用冒号 :，表示接下来是满足条件后要执行的语句块。
2. 使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。
3. 在 Python 中没有 switch...case 语句，但在 Python3.10 版本添加了 match...case，功能也类似，详见下文。

if 嵌套：在嵌套 if 语句中，可以把 if...elif...else 结构放在另外一个 if...elif...else 结构中。

#### match..case

Python 3.10 增加了 match...case 的条件判断，不需要再使用一连串的 if-else 来判断了。match 后的对象会依次与 case 后的内容进行匹配，如果匹配成功，则执行匹配到的表达式，否则直接跳过，\_ 可以匹配一切。

match subject:

    case <pattern\_1>:

        <action\_1>

    case <pattern\_2>:

        <action\_2>

    case <pattern\_3>:

        <action\_3>

    case \_:

        <action\_wildcard>

### 循环语句

Python 中的循环语句有 for 和 while。

#### while循环

一般形式：

while condition:

    # do something

* 无限循环：让条件一直为真（True）即可，可以使用 CTRL+C 退出无限循环。
* while…else：如果 while 后面的条件语句为 false 时，则执行 else 的语句块。

注意：while内通过break退出循环则不会执行else语句

#### for循环

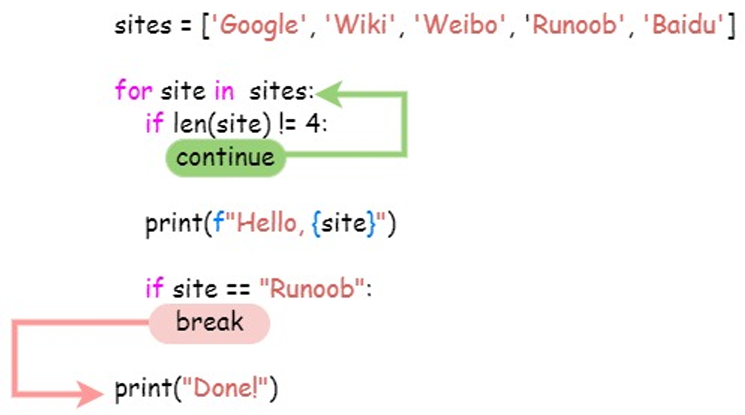
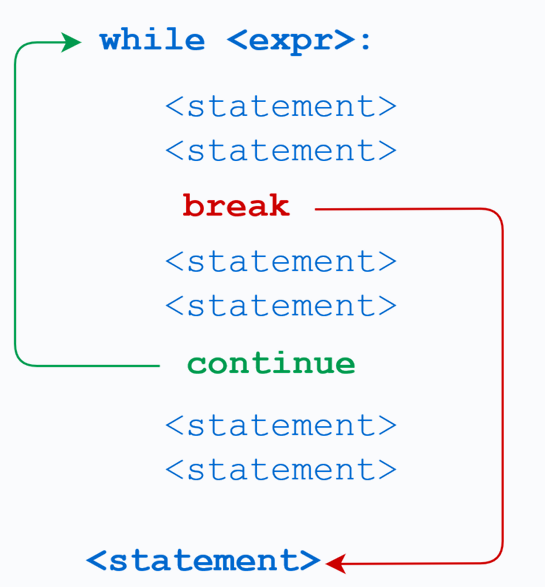
一般形式：

for <variable> in <sequence>:

<statements>

* for…else：类似地，在循环结束后执行代码，循环内遇到break也不会执行else语句。
* range(a,b,s)函数：获得a到b步长为s的数字列表，s可以为负数表示逆取；还可以通过\*解包组合多个range函数范围，例如(\*range(1,5),\*range(7,12))

#### break与continue



* break 语句可以跳出 for 和 while 的循环体。如果你从 for 或 while 循环中终止，任何对应的循环 else 块将不执行。
* continue 语句被用来告诉 Python 跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

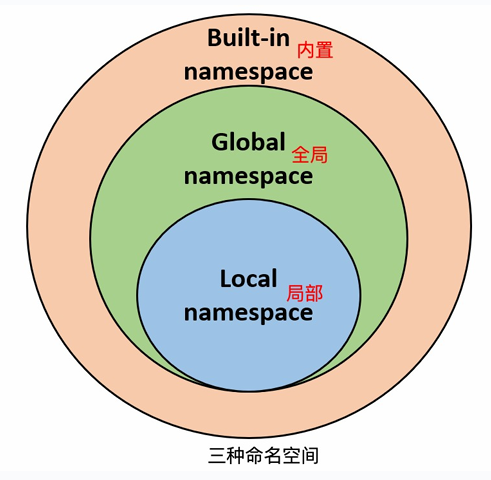
### 命名空间与作用域

#### 命名空间

* 命名空间(Namespace)是从名称到对象的映射，大部分的命名空间都是通过 Python 字典来实现的。
* 命名空间提供了在项目中避免名字冲突的一种方法。各个命名空间是独立的，没有任何关系的，所以一个命名空 间中不能有重名，但不同的命名空间是可以重名而没有任何影响。

##### 三种命名空间

* **内置名称（built-in names）**， **Python 语言内置**的名称，比如函数名 abs、char 和异常名称 BaseException、Exception 等等。
* **全局名称（global names）**，**模块**中定义的名称，记录了模块的变量，包括函数、类、其它导入的模块、模块级的变量和常量。
* **局部名称（local names）**，**函数**中定义的名称，记录了函数的变量，包括函数的参数和局部定义的变量。（类中定义的也是）



查找顺序：

由内到外：**局部的命名空间 -> 全局命名空间 -> 内置命名空间**。

##### 命名空间生命周期

命名空间的生命周期**取决于对象的作用域**，如果对象执行完成，则该命名空间的生命周期就结束。

无法从外部命名空间访问内部命名空间的对象，因为可能在外部命名空间时内部的命名空间生命周期已经结束导致无法访问。

#### 作用域

* 作用域就是一个 Python 程序可以直接访问命名空间的正文区域，定义了变量在哪些区域可以被访问。
* Python 采用 LEGB（Local、Enclosing、Global、Built-in）规则来决定变量查找的顺序。
* Python 中只有模块（module），类（class）以及函数（def、lambda）才会引入新的作用域，其它的代码块（如 if/elif/else/、try/except、for/while等）是不会引入新的作用域的，也就是说这些语句内定义的变量，外部也可以访问

##### LEGB作用域

* **L（Local）**：最内层，包含局部变量，比如一个函数/方法内部。
* **E（Enclosing）**：包含了非局部(non-local)也非全局(non-global)的变量。比如两个嵌套函数，一个函数（或类） A 里面又包含了一个函数 B ，那么对于 B 中的名称来说 A 中的作用域就为 nonlocal。
* **G（Global）**：当前脚本的最外层，比如当前模块的全局变量。
* **B（Built-in）**： 包含了内建的变量/关键字等，最后被搜索。

##### LEGB规则

**LEGB 规则**：Python 查找变量时的顺序是： L –> E –> G –> B

在局部找不到，便会去局部外的局部找（例如闭包），再找不到就会去全局找，再者去内置中找。



#### 全局变量和局部变量

* 定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。
* 局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。
* 在函数内部声明的变量只在函数内部的作用域中有效，调用函数时，这些内部变量会被加入到函数内部的作用域中，并且不会影响到函数外部的同名变量

total = 0 # 这是一个全局变量

# 可写函数说明

def sum( arg1, arg2 ):

    #返回2个参数的和."

    total = arg1 + arg2 # total在这里是局部变量.

    print ("函数内是局部变量 : ", total)

    return total

#调用sum函数

sum( 10, 20 )

 print ("函数外是全局变量 : ", total)

* 全局变量在函数外部定义，可以在整个文件中访问。
* 局部变量在函数内部定义，只能在函数内访问。
* 使用 global 可以在函数中修改全局变量。
* 使用 nonlocal 可以在嵌套函数中修改外部函数的变量。

##### 全局变量

**全局变量**：在函数外部定义的变量，可以在整个文件中被访问。在函数外部定义的变量对所有函数都是可见 的，除非函数内部定义了同名的局部变量。

##### 局部变量

**局部变量**：在函数内部定义的变量，仅在函数内有效，函数外无法访问。局部变量优先级高于全局变量，因 此如果局部变量和全局变量同名，函数内部会使用局部变量。

##### global 和 nonlocal关键字

当内部作用域想修改外部作用域的变量时，就要用到 global 和 nonlocal 关键字

## Python数据结构

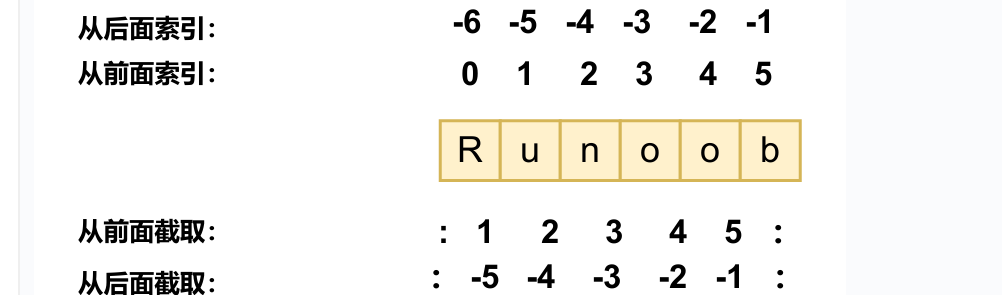
### 字符串

字符串是 Python 中最常用的数据类型，可以使用引号( ' 或 " )来创建字符串。单双引号本质上没有区别，且特定情况可以结合使用划分范围。

#### 访问字符串

单字符在 Python 中也是作为一个字符串使用。

Python 使用方括号 [] 来截取字符串访问子字符串，如：变量[头下标:尾下标]



#### 字符串转义

在需要在字符中使用特殊字符时，python 用反斜杠 \ 转义字符。如下表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| \(在行尾时) | 续行 | \b | Backspsce | \r | 回车 |
| \\ | 反斜杠符号 | \000 | 空 | \f | 换页 |
| \’ | 单引号 | \n | 换行 | \yyy | 按八进制数 |
| \” | 双引号 | \v | 纵向制表符 | \xyy | 按十六进制 |
| \a | 响铃 | \t | 横向制表符 | \other | 普通格式 |

使用 \r 实现百分比进度

for i in range(101):

    print("\r{:3}%".format(i),end=' ')

    time.sleep(0.05)

#### 字符串运算

字符串间可以进行运算操作，入下表

|  |  |
| --- | --- |
| + | 字符串连接 |
| \* | 重复输出字符串 |
| [] | 通过索引获取字符串中字符（类似列表） |
| [:] | 截取字符串中的一部分，遵循左闭右开原则 |
| in | 成员运算符 - 如果字符串中包含给定的字符返回 True |
| not in | 成员运算符 - 如果字符串中不包含给定的字符返回 True |
| r/R | 原始字符串 - 原始字符串：r’字符串’ |
| % | 格式字符串 |

#### 格式字符串

Python 支持格式化字符串的输出 。最基本的用法是将一个值插入到一个有字符串格式符 %s 的字符串中。字符串格式化使用与 C 中 sprintf 函数一样的语法。例如：

print ("我叫 %s 今年 %d 岁!" % ('小明', 10))

有下表

|  |  |  |  |
| --- | --- | --- | --- |
| %c | 格式化字符及其ASCII码 | %X | 格式化无符号十六进制数（大写） |
| %s | 格式化字符串 | %e/%E | 用科学计数法格式化浮点数 |
| %d | 格式化整数 | %g | %f和%e的简写 |
| %u | 格式化无符号整型 | %G | %f 和 %E 的简写 |
| %o | 格式化无符号八进制数 | %p | 用十六进制数格式化变量的地址 |
| %x | 格式化无符号十六进制数 | %f | 格式化浮点数字，可指定精度 |

格式化操作符辅助指令:

|  |  |
| --- | --- |
| \* | 定义宽度或者小数点精度 |
| - | 用做左对齐 |
| + | 在正数前面显示加号( + ) |
| <sp> | 在正数前面显示空格 |
| # | 在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X') |
| 0 | 显示的数字前面填充'0'而不是默认的空格 |
| % | '%%'输出一个单一的'%' |
| (var) | 映射变量(字典参数) |
| m.n | m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话) |

#### 三引号

三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。

#### f-string

字面量格式化字符串，f-string 格式化字符串以 f 开头，后面跟着字符串，字符串中的表达式用大括号 {} 包起来，它会将变量或表达式 计算后的值替换进去。

详细可见8.1.1.4.2.输出

### 列表

* 列表是最常用的 Python 数据类型，它可以作为一个方括号内的逗号分隔值出现。
* 列表的数据项不需要具有相同的类型

list1 = ['Google', 'Runoob', 1997, 2000]

list2 = [1, 2, 3, 4, 5 ]

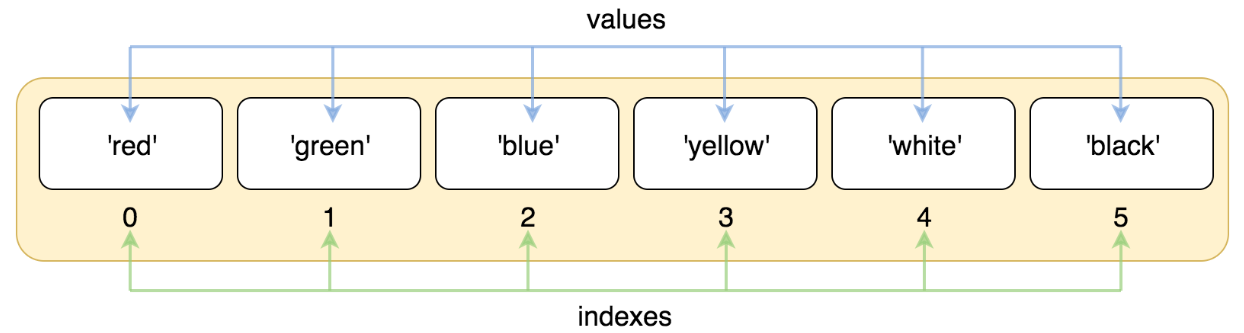
list3 = ["a", "b", "c", "d"]

list4 = ['red', 'green', 'blue', 'yellow', 'white', 'black']

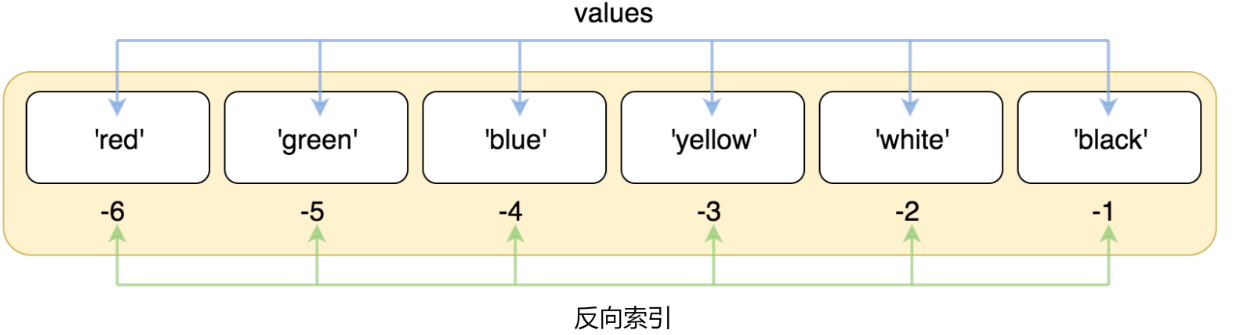
* 序列中的每个值都有对应的位置值，称之为索引，第一个索引是 0，第二个索引是 1，依此类推。
* 列表都可以进行的操作包括索引，切片，加，乘，检查成员。

#### 访问列表

与字符串的索引一样，列表索引从 0 开始，第二个索引是 1，依此类推；



索引也可以从尾部开始，最后一个元素的索引为 -1，往前一位为 -2，以此类推；



下标索引来访问列表中的值，同样你也可以使用方括号 [] 的形式截取字符

#### 更新列表

列表属于可变数据类型，可以对列表的数据项进行修改或更新，也可以使用 append() 方法来添加列表项：可以通过[]访问列表元素进行列表内容的修改。

list = ['Google', 'Runoob', 1997, 2000]

print ("第三个元素为 : ", list[2])  # 输出第三个元素#: 1997

list[2] = 2001

print ("更新后的第三个元素为 : ", list[2])  # 输出更新后的第三个元素#: 2001

#### 删除列表

以使用 del 语句来删除列表中的元素。

list = ['Google', 'Runoob', 1997, 2000]

print ("原始列表 : ", list)     #['Google', 'Runoob', 1997, 2000]

del list[2]

print ("删除第三个元素 : ", list)   #['Google', 'Runoob', 2000]

#### 列表操作符

|  |  |  |
| --- | --- | --- |
| len([1, 2, 3]) | 3 | 长度 |
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6] | 组合 |
| ['Hi!'] \* 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | 重复 |
| 3 in [1, 2, 3] | True | 元素是否存在于列表中 |
| for x in [1, 2, 3]:  print(x, end=" ") | 1 2 3 | 迭代 |

#### 列表截取与拼接

Python 的列表截取与字符串操作类似。

#### 嵌套列表

嵌套列表即在列表里创建其它列表，例如

[['a', 'b', 'c'], [1, 2, 3]]

#### 列表函数方法

|  |  |
| --- | --- |
| list.append(obj)  在列表末尾添加新的对象 | list.remove(obj)  移除列表中某个值的第一个匹配项 |
| list.count(obj)  统计某个元素在列表中出现的次数 | list.reverse()  反向列表中元素 |
| list.extend(seq)  在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表） | list.sort( key=None, reverse=False)  对原列表进行排序 |
| list.index(obj)  从列表中找出某个值第一个匹配项的索引位置 | list.clear()  清空列表 |
| list.insert(index, obj)  将对象插入列表 | list.copy()  复制列表 |
| list.pop([index=-1])  移除列表中的一个元素（默认最后一个元素），并且返回该元素的值 |  |

### 元组

* 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 = ('Google', 'Runoob', 1997, 2000)

* 元组中只包含一个元素时，需要在元素后面添加逗号’,’，否则括号会被当作运算符使用。
* 元组与字符串类似，下标索引从 0 开始，可以进行截取，组合等。

#### 访问元组

类似地，元组可以使用下标索引来访问元组中的值，如下实例

tup1 = ('Google', 'Runoob', 1997, 2000)

tup2 = (1, 2, 3, 4, 5, 6, 7 )

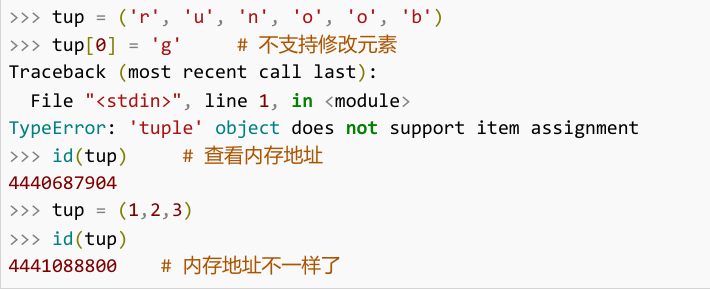
print ("tup1[0]: ", tup1[0])        # tup1[0]:  Google

print ("tup2[1:5]: ", tup2[1:5])    # tup2[1:5]:  (2, 3, 4, 5)

#### 修改/删除元组

元组为不可变数据类型，其中的元素值是不允许修改的。

所谓元组的不可变指的是元组所指向的内存中的内容不可变。



重新赋值的元组 tup，绑定到新的对象了，不是修改了原来的对象。

但我们可以对元组进行连接组合

* 元组组合

tup1 = (12, 34.56)

tup2 = ('abc', 'xyz')

tup3 = tup1 + tup2

print (tup3)    # Output: (12, 34.56, 'abc', 'xyz')

* 元组删除：元组中的元素值是不允许删除的，但我们可以使用del语句来删除整个元组

#### 元组运算

元组之间可以使用 +、+=和 \* 号进行运算。这就意味着他们可以组合和复制，运算后会生成新的元组。

#### 元组索引与截取

元组也是一个序列，所以我们可以访问元组中的指定位置的元素，也可以截取索引中的一段元素。同样的[]、[:]都可以使用。

#### 元组内置函数

|  |  |
| --- | --- |
| len(tuple)  计算元组元素个数。 | min(tuple)  返回元组中元素最小值。 |
| max(tuple)  返回元组中元素最大值。 | tuple(iterable)  将可迭代系列转换为元组。 |

### 字典

* 字典是另一种可变容器模型，且可存储任意类型对象。
* 字典的每个键值 key=>value 对用冒号 : 分割，每个对之间用逗号(,)分割，整个字典包括在花括号 {} 中 ,格式如下：

d = {key1 : value1, key2 : value2, key3 : value3 }

* 键必须是唯一的，但值则不必

#### 创建空字典

* 使用大括号 { } 创建空字典

emptyDict = {}

* 使用内建函数 dict() 创建字典

emptyDict = dict()

#### 访问字典

通过下面格式访问字典：字典[键值]

tinydict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}

print ("tinydict['Name']: ", tinydict['Name'])  #tinydict['Name']:  Runoob

print ("tinydict['Age']: ", tinydict['Age'])    # tinydict['Age']:  7

#### 修改字典

* 修改字典：访问到字典后赋值
* 添加字典元素：为先前没有的键赋值

tinydict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}

tinydict['Age'] = 8               # 更新 Age

tinydict['School'] = "菜鸟教程"  # 添加信息

print ("tinydict['Age']: ", tinydict['Age'])    #8

print ("tinydict['School']: ", tinydict['School'])  #菜鸟教程

#### 删除字典元素

* 使用 del 语句删除字典中的元素或整个字典

del dict[key]   #删除对应键值对

del dict    #删除整个字典

* 使用 pop() 方法删除字典中的元素，并返回该元素的值。

value = dict.pop(key, default\_value)

* 使用 popitem() 方法删除字典中的最后一个键值对

key, value = dict.popitem()

* 使用 clear() 方法清空字典

dict.clear()

#### 字典键的特性

字典值可以是任何的 python 对象，既可以是标准的对象，也可以是用户定义的，但键不行。

* 不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住
* 键必须不可变，所以可以用数字，字符串或元组充当，而用列表就不行

#### 字典内置函数方法

* 内置函数

|  |  |  |
| --- | --- | --- |
| len(dict)  计算字典元素个数，即键 的总数。 | str(dict)  输出字典，可以打印的字 符串表示。 | type(variable)  返回输入的变量类型，如 果变量是字典就返回字典 类型。 |

* 内置方法

|  |  |
| --- | --- |
| dict.clear()  删除字典内所有元素 | dict.keys()  返回一个视图对象 |
| dict.copy()  返回一个字典的浅复制 | dict.setdefault(key, default=None)  和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default |
| dict.fromkeys()  创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值 | dict.update(dict2)  把字典dict2的键/值对更新到dict里 |
| dict.get(key, default=None)  返回指定键的值，如果键不在字典中返回 default 设置的默认值 | dict.values()  返回一个视图对象 |
| key in dict  如果键在字典dict里返回true，否则返回false | dict.pop(key[,default])  删除字典 key（键）所对应的值，返回被删除的值。 |
| dict.items()  以列表返回一个视图对象 | dict.popitem()  返回并删除字典中的最后一对键和值。 |

### 集合

* 集合（set）是一个无序的不重复元素序列。
* 集合中的元素不会重复，并且可以进行交集、并集、差集等常见的集合操作。
* 可以使用大括号 { } 创建集合，元素之间用逗号分隔或者也可以使用 set() 函数创建集合。创建格式：

parame = {value01,value02,...}

set(value)

注意：创建一个空集合必须用 set() 而不是 { }，因为 { } 是用来创建一个空字典。

集合内置函数方法如下：

|  |  |
| --- | --- |
| add() | 为集合添加元素 |
| clear() | 移除集合中的所有元素 |
| copy() | 拷贝一个集合 |
| difference() | 返回多个集合的差集 |
| difference\_update() | 删除集合中指定的元素 |
| discard() | 返回集合的交集。 |
| intersection() | 集合的交集。 |
| intersection\_update() | 更新原集合，使其只包含交集部分 |
| isdisjoint() | 判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False |
| issubset() | 判断指定集合是否为该方法参数集合的子集。 |
| issuperset() | 判断该方法的参数集合是否为指定集合的子集 |
| pop() | 随机移除元素 |
| remove() | 移除指定元素 |
| symmetric\_difference() | 返回两个集合中不重复的元素集合。 |
| symmetric\_difference\_update() | 移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不 同的元素插入到当前集合中。 |
| union() | 返回两个集合的并集 |
| update() | 给集合添加元素 |
| len() | 计算集合元素个数 |

### 推导式

推导式是一种独特的数据处理方式，可以从一个数据序列构建另一个新的数据序列的结构体，是一种强大且简洁的语法，适用于生成列表、字典、集合和生成器。

使用推导式时，需要注意可读性，保持表达式简洁，以免影响代码的可读性和可维护性。

Python 支持各种数据结构的推导式：

1. 列表(list)推导式
2. 字典(dict)推导式
3. 集合(set)推导式
4. 元组(tuple)推导式

### 列表推导式

[表达式 for 变量 in 列表]

[out\_exp\_res for out\_exp in input\_list]

或者

[表达式 for 变量 in 列表 if 条件]

[out\_exp\_res for out\_exp in input\_list if condition]

* out\_exp\_res：列表生成元素表达式，可以是有返回值的函数。
* for out\_exp in input\_list：迭代 input\_list 将 out\_exp 传入到 out\_exp\_res 表达式中。
* if condition：条件语句，可以过滤列表中不符合条件的值。

### 字典推导式

{ key\_expr: value\_expr for value in collection }

或

{ key\_expr: value\_expr for value in collection if condition }

例如：

listdemo = ['Google','Runoob', 'Taobao']

# 将列表中各字符串值为键，各字符串的长度为值，组成键值对

newdict = {key:len(key) for key in listdemo}

# {'Google': 6, 'Runoob': 6, 'Taobao': 6}

### 集合推导式

{ expression for item in Sequence }

或

{ expression for item in Sequence if conditional }

例如：

setnew = {i\*\*2 for i in (1,2,3)}

#setnew= {1, 4, 9}

### 元组推导式（生成器表达式）

元组推导式可以利用 range 区间、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的元组。

(expression for item in Sequence )

或

(expression for item in Sequence if conditional )

元组推导式和列表推导式的用法也完全相同，只是元组推导式是用 () 圆括号将各部分括起来，而列表推导式用的是中括号 []，另外元组推导式返回的结果是一个生成器对象。

### 其它数据结构

#### 栈

Python 中不直接提供栈这种数据结构，但可以使用列表（list）来实现栈的功能。栈是一种后进先出（LIFO, Last-In-First-Out）数据结构， 意味着最后添加的元素最先被移除。列表提供了一些方法，使其非常适合用于栈操作，特别是 append() 和 pop() 方法。

栈操作：

* 压入（Push）: 将一个元素添加到栈的顶端。
* 弹出（Pop）: 移除并返回栈顶元素。
* 查看栈顶元素（Peek/Top）: 返回栈顶元素而不移除它。
* 检查是否为空（IsEmpty）: 检查栈是否为空。
* 获取栈的大小（Size）: 获取栈中元素的数量。

以下是一个完整的实例，展示了如何使用列表来实现一个简单的栈：

class Stack:

    def \_\_init\_\_(self):

        self.stack = []

    def push(self, item):

        self.stack.append(item)

    def pop(self):

        if not self.is\_empty():

            return self.stack.pop()

        else:

            raise IndexError("pop from empty stack")

    def peek(self):

        if not self.is\_empty():

            return self.stack[-1]

        else:

            raise IndexError("peek from empty stack")

    def is\_empty(self):

        return len(self.stack) == 0

    def size(self):

        return len(self.stack)

 # 使用示例

stack = Stack()

stack.push(1)

stack.push(2)

stack.push(3)

print("栈顶元素:", stack.peek())  # 输出: 栈顶元素: 3

print("栈大小:", stack.size())    # 输出: 栈大小: 3

print("弹出元素:", stack.pop())  # 输出: 弹出元素: 3

print("栈是否为空:", stack.is\_empty())  # 输出: 栈是否为空: False

print("栈大小:", stack.size())    # 输出: 栈大小: 2

#### 队列

在 Python 中，也不提供队列，但列表（list）可以用作队列（queue），但由于列表的特点，直接使用列表来实现队列并不是最优的选择。队列是一种先进先出（FIFO, First-In-First-Out）的数据结构，意味着最早添加的元素最先被移除。

Python 提供了 collections.deque，它是双端队列，可以在两端高效地添加和删除元素。

class Queue:

    def \_\_init\_\_(self):

        self.queue = []

    def enqueue(self, item):

        self.queue.append(item)

    def dequeue(self):

        if not self.is\_empty():

            return self.queue.pop(0)

        else:

            raise IndexError("dequeue from empty queue")

    def peek(self):

        if not self.is\_empty():

            return self.queue[0]

        else:

            raise IndexError("peek from empty queue")

    def is\_empty(self):

        return len(self.queue) == 0

    def size(self):

        return len(self.queue)

 # 使用示例

queue = Queue()

queue.enqueue('a')

queue.enqueue('b')

queue.enqueue('c')

print("队首元素:", queue.peek())    # 输出: 队首元素: a

print("队列大小:", queue.size())    # 输出: 队列大小: 3

print("移除的元素:", queue.dequeue())  # 输出: 移除的元素: a

print("队列是否为空:", queue.is\_empty())  # 输出: 队列是否为空: False

print("队列大小:", queue.size())    # 输出: 队列大小: 2

## Python函数与模块

### 函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

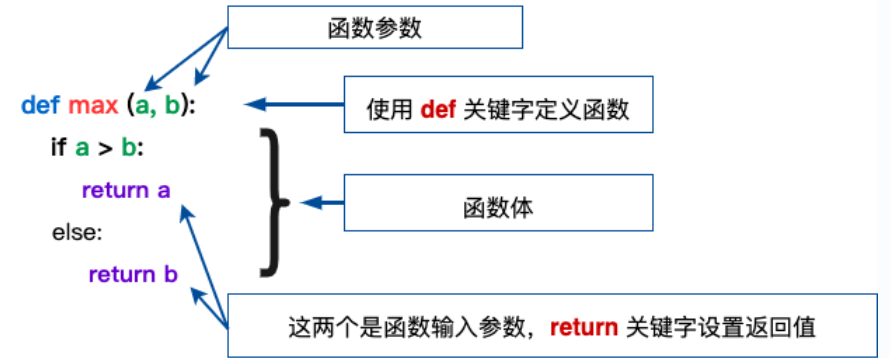
函数能提高应用的模块性，和代码的重复利用率。

#### 定义函数

**定义函数**是指在编程中通过一个特定的语法为某个特定任务或功能创建一个可复用的代码块。定义函数时，你指定函数的名称、参数（可选）以及函数体（即函数的具体实现）。

以下是简单的规则：

* 函数代码块以 def 关键词开头，后接函数标识符名称和圆括号 ()。
* 任何传入参数和自变量必须放在圆括号中间，圆括号之间可以用于定义参数。
* 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。
* 函数内容以冒号 : 起始，并且缩进。
* return [表达式] 结束函数，选择性地返回一个值给调用方，不带表达式的 return 相当于返回 None。



#### 函数调用

**调用函数**是指在程序中使用已经定义好的函数，以便执行该函数的功能。通过调用函数，你可以让函数执行其内部的代码，处理传入的参数（如果有），并返回一个结果（如果函数有返回值）。

#### 参数传递之对象传递方式

python 中，类型是对象的分类，变量是没有类型：

* 每个变量都是指向一个对象的引用。
* 对象的类型决定了该对象可以执行哪些操作。
* 变量只是对象的一个名字或引用，多个变量可以指向同一个对象

a=[1,2,3]

a="Runoob"

以上，[1,2,3] 是 List 类型，"Runoob" 是 String 类型，而变量 a 是没有类型，它仅仅是一个对象的引用（一个指针），可以是指向 List 类型对象，也可以是指向 String 类型对象。

可变对象是可以修改其内容的对象，而不可变对象则是无法改变其内容的对象。

* **可变对象**是指其内容可以在对象创建之后被修改的对象。这意味着，给可变对象赋予新值时，内存中的对象本身并没有改变，而是直接修改了对象的内容。

**例如：**列表、字典、集合

* **不可变对象**是指一旦对象被创建，它的内容就不能修改的对象。这意味着，如果你尝试修改不可变对象的内容，实际上会创建一个新的对象，而不是修改原始对象。

**例如：**整数、浮点数、字符串、元组、冻结集合

参数传递时，可以这样理解：可变对象为引用传递，不可变对象为赋值传递，详细如后。

* 引用传参（Pass by Reference）：传递本身，内外共用
* 赋值传参（Pass by Assignment）：传递一个副本，变量内部修改不会影响外部

**注意：python 中一切都是对象，严格意义我们不能说值传递还是引用传递，我们应该说传不可变对象和传可变对象。**

#### 参数传递之参数类型

是调用函数时可使用的正式参数类型：必需参数、关键字参数、默认参数、不定长参数

* **必需参数**

1. 必需参数须以正确的顺序传入函数。
2. 调用时的数量必须和声明时的一样。

* **关键字参数**

1. 关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。
2. 使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值

* **默认参数**

定义是设置默认参数，调用函数时，如果没有传递参数，则会使用默认参数

* **不定长参数**

需要一个函数能处理比当初声明时更多的参数时，定义的参数叫做不定长参数，声明时 不会命名。基本语法如下：

def functionname([formal\_args,] \*var\_args\_tuple ):

    function\_suite

    return [expression]

星号 \* 的参数会以元组(tuple)的形式导入，存放所有未命名的变量参数。而\*\*则会以字典形式导入，这种方法是更推荐的，因为以字典导入会根据字典键对应关键字来取值。

**,\*,**：当函数定义是出现单独的\*（星号）时，表明该\*后的参数必须使用关键字传参（指定关键字 = 值）

/：函数形参语法/用来指明函数形参必须使用指定位置参数，不能使用关键字参数的形式。

def f(a, b, /, c, d, \*, e, f):

    print(a, b, c, d, e, f)

f(10, 20, 30, d=40, e=50, f=60)     # 正确调用

f(10, b=20, c=30, d=40, e=50, f=60)   # b 不能使用关键字参数的形式

f(10, 20, 30, 40, 50, f=60)           # e 必须使用关键字参数的形式

#### 匿名函数lambda

* Python 使用 lambda 来创建匿名函数。lambda 函数是一种小型、匿名的、内联函数，它可以具有任意数量的参数，但只能有一个表达式。
* 匿名函数不需要使用 def 关键字定义完整函数。
* lambda 函数通常用于编写简单的、单行的函数，通常在需要函数作为参数传递的情况下使用，例如在 map()、filter()、reduce() 等函数中。
* lambda 函数特点：

1. lambda 函数是匿名的，它们没有函数名称，只能通过赋值给变量或作为参数传递给其他函数来使用。
2. lambda 函数通常只包含一行代码，这使得它们适用于编写简单的函数。

* lambda 语法格式：

lambda arguments: expression

1. lambda是 Python 的关键字，用于定义 lambda 函数。
2. arguments 是参数列表，可以包含**零个或多个**参数，但必须在冒号(:)前指定。
3. expression 是一个表达式，用于计算并返回函数的结果。

匿名函数接收零或多个参数：

#无参数

f = lambda: "Hello, world!"

print(f())  # 输出: Hello, world!

#一个参数

x = lambda a : a + 10

print(x(5)) # 输出: 15

#多个参数

x = lambda a, b : a \* b

print(x(5, 6))  # 输出: 30

#### 其它特殊函数

* 内置函数：如 print(), len(), max() 等。

def factorial(n):

    if n == 0:

        return 1  # 基准情况

    else:

        return n \* factorial(n-1)  # 递归调用

* 魔术方法（特殊方法）：如 \_\_init\_\_()、\_\_str\_\_()、\_\_call\_\_() 等，用于实现类的特殊行为。
* 生成器函数：使用 yield 关键字定义，生成一个迭代器。
* 装饰器函数：用于修改或增强其他函数或方法的功能。

### 模块

Python 中的模块（Module）是一个包含 Python 定义和语句的文件，文件名就是模块名加上 .py 后缀。模块可以包含函数、类、变量以及可执行的代码。通过模块，我们可以将代码组织成可重用的单元，便于管理和维护。

**模块的作用**

* 代码复用：将常用的功能封装到模块中，可以在多个程序中重复使用。
* 命名空间管理：模块可以避免命名冲突，不同模块中的同名函数或变量不会互相干扰。
* 代码组织：将代码按功能划分到不同的模块中，使程序结构更清晰。

#### 模块导入

* **import 语句：**
* 使用 Python 源文件，只需在另一个源文件里执行 import 语句。
* 一个模块只会被导入一次，不管你执行了多少次 import。
* 当导入一个模块时，Python 会按照以下顺序查找模块：

1. 当前目录。
2. 环境变量 PYTHONPATH 指定的目录。
3. Python 标准库目录。
4. .pth 文件中指定的目录。

* **from … import 语句：**
  + 的 from 语句允许从模块中导入一个指定的部分到当前命名空间中。
  + from … import \*：把一个模块的所有内容全都导入到当前的命名空间，但是不推荐，容易引起命名冲突。
* **as：**

用于为模块或函数起别名

#### \_\_name\_\_ 属性

一个模块被另一个程序第一次引入时，其主程序将运行。

如果我们想在模块被引入时，模块中的某一程序块不执行，我们可以用 \_\_name\_\_ 属性来使该程序块仅在该模块 自身运行时执行。



说明：每个模块都有一个 \_\_name\_\_ 属性。

* 如果模块是被直接运行，\_\_name\_\_ 的值为 \_\_main\_\_。
* 如果模块是被导入的，\_\_name\_\_ 的值为模块名。

#### 标准模块

Python 本身带着一些标准的模块库。

|  |  |
| --- | --- |
| math | 数学运算（如平方根、三角函数等） |
| os | 操作系统相关功能（如文件、目录操作） |
| sys | 系统相关的参数和函数 |
| random | 生成随机数 |
| datetime | 处理日期和时间 |
| json | 处理 JSON 数据 |
| re | 正则表达式操作 |
| collections | 提供额外的数据结构（如 defaultdict、deque） |
| itertools | 提供迭代器工具 |
| functools | 高阶函数工具（如 reduce、lru\_cache） |

#### 包

* 包是一种管理 Python 模块命名空间的形式，采用"点模块名称"。比如一个模块的名称是 A.B， 那么他表示一个包 A中的子模块 B 。
* 就好像使用模块的时候，你不用担心不同模块之间的全局变量相互影响一样，采用点模块名称这种形式也不用担心不同库之间的模块重名的情况。
* 导入一个包的时候，Python 会根据 sys.path 中的目录来寻找这个包中包含的子目录。 目录只有包含一个叫做 \_\_init\_\_.py 的文件才会被认作是一个包。

一个包实际上是一个目录，目录中必须包含一个名为 \_\_init\_\_.py 的文件。这个文件可以为空，或者包含包的初始化代码。基本结构如下：

mypackage/                   # 包的目录

    \_\_init\_\_.py              # 包的初始化文件

    module1.py               # 包内的一个模块

    module2.py               # 包内的另一个模块

    subpackage/              # 子包

        \_\_init\_\_.py          # 子包的初始化文件

        submodule.py         # 子包内的模块

## Python面向对象OOP

### 面向对象简介

面向对象编程（Object-Oriented Programming, OOP）是一种组织程序结构的方式，它通过使用类和对象来将数据和操作数据的方法封装在一起。Python 支持面向对象的编程方式，它提供了丰富的特性来实现对象的创建、继承、封装、多态和抽象等特性。

核心概念：

* **类(Class):** 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
* **方法：**类中定义的函数。
* **类变量：**类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
* **数据成员：**类变量或者实例变量用于处理类及其实例对象的相关的数据。
* **方法重写：**如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重写。
* **局部变量：**定义在方法中的变量，只作用于当前实例的类。
* **实例变量：**在类的声明中，属性是用变量来表示的，这种变量就称为实例变量，实例变量就是一个用 self 修饰的变量。
* **继承：**即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个Dog类型的对象派生自Animal类，这是模拟"是一个（is-a）"关系（例图，Dog是一个Animal）。
* **实例化：**创建一个类的实例，类的具体对象。
* **对象：**通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。

### 类和对象

#### 类

类是创建对象的模板，它定义了对象的属性（变量）和行为（方法）。类本身是一个数据类型，可以用来生成实例（对象）。

class Dog:

    # 类属性

    species = "Canis familiaris"

    # 构造方法（初始化方法）

    def \_\_init\_\_(self, name, age):

        self.name = name  # 实例属性

        self.age = age    # 实例属性

    # 类的方法

    def bark(self):

        print(f"{self.name} says Woof!")

#### 对象

对象是类的实例化。通过类创建对象时，类的构造方法 \_\_init\_\_ 会被调用。

dog1 = Dog("Buddy", 5)  # 创建对象

dog1.bark()  # 调用方法

### 类方法

* 在类的内部，使用 **def** 关键字来定义一个方法，与一般函数定义不同，类方法必须包含参数 self, 且为第一个参数，self 代表的是类的实例。
* 类的方法定义了对象的行为。方法是与类或对象相关联的函数。

#### 构造方法\_\_init\_\_

\_\_init\_\_ 是类的构造方法，通常用于初始化对象的属性。当你创建一个对象时，\_\_init\_\_ 会被自动调用。

在实例化类对象时，需要传入构造方法中定义的参数来创建类实例。

class Car:

    def \_\_init\_\_(self, brand, model):

        self.brand = brand

        self.model = model

car1 = Car("Toyota", "Corolla")

print(car1.brand)  # 输出 "Toyota"

#### 静态方法与类方法

* **静态方法（@staticmethod）**：与类和实例无关的方法，它不能访问实例或类的属性。它只关注传入的参数。

class Math:

    @staticmethod

    def add(x, y):

        return x + y

* **类方法（@classmethod）**：类方法的第一个参数是类本身（cls），可以访问类的属性，但不能访问实例的属性。

class MyClass:

    count = 0

    @classmethod

    def increment\_count(cls):

        cls.count += 1

        print(cls.count)

MyClass.increment\_count()  # 通过类调用

### 类中的属性

* **实例属性**：实例化对象后，属于该对象的属性。
* **类属性**：属于类的属性，而不是某个特定对象的属性。所有实例都共享类属性。

class Dog:

    species = "Canis familiaris"  # 类属性

    def \_\_init\_\_(self, name, age):

        self.name = name  # 实例属性

        self.age = age    # 实例属性

### 继承

继承是面向对象的核心特性之一，子类可以继承父类的属性和方法，从而实现代码复用。子类还可以重写父类的方法，或者添加新的方法。

#类定义

class people:

    #定义基本属性

    name = ''

    age = 0

    #定义私有属性,私有属性在类外部无法直接进行访问

    \_\_weight = 0

    #定义构造方法

    def \_\_init\_\_(self,n,a,w):

        self.name = n

        self.age = a

        self.\_\_weight = w

    def speak(self):

        print("%s 说: 我 %d 岁。" %(self.name,self.age))

#单继承示例

class student(people):

    grade = ''

    def \_\_init\_\_(self,n,a,w,g):

        #调用父类的构函

        people.\_\_init\_\_(self,n,a,w)

        self.grade = g

    #覆写父类的方法

    def speak(self):

        print("%s 说: 我 %d 岁了，我在读 %d 年级"%(self.name,self.age,self.grade))

s = student('ken',10,60,3)

s.speak()

#### 多继承

多继承（Multiple Inheritance）​是指一个子类可以同时继承多个父类的属性和方法，从而实现代码的灵活复用和功能组合。

**方法解析顺序：**

1. ​**子类优先级高于父类**：例如子类重写父类方法时优先调用自身实现。
2. ​**继承列表顺序从左到右**：父类列表左侧的类优先级更高。
3. ​**单调性原则**：保证继承链的线性顺序，避免循环调用

#### 重写与多态

* 子类可以**重写**父类的方法。通过方法重写，子类可以提供自己的实现。
* 多态指的是同一个方法或操作在不同的对象上有不同的行为。通过**继承和方法重写**可以实现多态。

class Animal:

    def sound(self):

        print("Some generic sound")

class Dog:

    def sound(self):

        print("Woof!")  #重写

class Cat:

    def sound(self):

        print("Meow!")  ##重写

def make\_sound(animal):

    animal.sound()

dog = Dog()

cat = Cat()

make\_sound(dog)  # 输出 "Woof!"

make\_sound(cat)  # 输出 "Meow!"

### 封装

封装是将对象的状态（属性）和行为（方法）打包成一个单独的单位，隐藏对象的内部实现细节，只暴露必要的接口。Python 使用类来实现封装。

* 公有成员（Public）：类中的普通属性和方法，外部可以访问。
* 私有成员（Private）：python不存在public、private等关键字，通过在属性或方法名前加双下划线（\_\_）来实现，外部不能直接访问。

class Dog:

    def \_\_init\_\_(self, name, age):

        self.name = name        # 公有属性

        self.\_\_age = age        # 私有属性

    def get\_age(self):         # 公有方法

        return self.\_\_age

    def set\_age(self, age):    # 公有方法

        if age > 0:

            self.\_\_age = age

d = Dog("Buddy", 5)

print(d.name)      # 可以访问公有属性

# print(d.\_\_age)   # 会抛出 AttributeError，无法访问私有属性

print(d.get\_age())  # 通过公有方法访问私有属性

### 抽象

抽象类是不能实例化的类，它主要用于作为其他类的基类。通过抽象类，其核心作用是定义接口规范并强制子类实现特定方法。Python 提供了 abc 模块来支持抽象类。

* 子类直接继承抽象父类且未被标记为抽象类时，必须实现父类中所有用 @abstractmethod 装饰的抽象方法。
* 若子类本身也声明为抽象类（继承 ABC 或使用 @abstractmethod），则可以不实现父类的抽象方法，将实现责任传递给下一层子类

## Python错误与异常处理

### 区分错误与异常

* **错误（Error）**：表示程序中无法继续执行的严重问题。通常，错误无法被程序捕获和处理，只能通过修复代码来解决。常见的错误包括语法错误（SyntaxError）等。
* **异常（Exception）**：是程序运行时可能发生的错误，它可以被程序捕获并处理，从而使程序继续执行。常见的异常包括索引越界（IndexError）、文件未找到（FileNotFoundError）等。

简单区分：

错误发生在运行前，异常出现在运行中；前者为编码问题，后者为运行时特殊情况导致，可以被捕获。

### 异常处理

异常捕捉可以使用 try/except/else/finally 语句。

* try: 用于放置可能会引发异常的代码
* except: 用于捕获异常并处理。可以捕获特定类型的异常，也可以捕获所有异常。
* else: 如果没有发生异常,正常结束，则执行 else 中的代码（可选）。
* finally: 无论是否发生异常，都会执行 finally 中的代码（可选），通常用于清理资源，如关闭文件或网络连接。



例：

try:

    x = 10 / 0  # 会引发 ZeroDivisionError

except ZeroDivisionError as e:

    print("除以零错误:", e)

else:

    print("没有异常")

finally:

print("无论如何都会执行")

通过多个except语句可以按顺序捕获多个异常

### 常见异常

|  |  |
| --- | --- |
| IndexError | 索引越界错误，当访问一个不存在的列表或元组元素时抛出。 |
| ValueError | 值错误，当一个操作或函数接收到一个具有正确类型但不适当值的参数时抛出。 |
| TypeError | 类型错误，当发生不支持的操作时抛出，比如对不同类型的数据进行不兼容操作。 |
| KeyErrorv | 字典访问时，键不存在抛出此异常。 |
| FileNotFoundError | 文件未找到错误，当打开一个不存在的文件时抛出。 |
| ZeroDivisionError | 除以零错误 |
| AttributeError | 当对象没有这个属性时抛出 |
| ImportError | 当导入模块失败时抛出 |

### 自定义异常

通过继承内置的 Exception 类来自定义异常类，用raise来触发异常

class MyError(Exception):

    def \_\_init\_\_(self, value):

        self.value = value

    def \_\_str\_\_(self):

        return repr(self.value)

#case1

try:

    raise MyError(2\*2)

except MyError as e:

    print('My exception occurred, value:', e.value)

# Output: My exception occurred, value: 4

#case2

raise MyError('oops!')

#Traceback (most recent call last):

#  File "<stdin>", line 1, in ?

#\_\_main\_\_.MyError: 'oops!'

上面例子中：

* case1捕获到了异常，因此可以输出except内的内容。
* case2没有捕获，直接程序终止而报错

### 断言assert

* assert（断言）用于判断一个表达式，在表达式条件为 false 的时候触发异常。
* 断言可以在条件不满足程序运行的情况下直接返回错误，而不必等待程序运行后出现崩溃的情况。
* 语法格式如下：[,message]表示可选参数，用于提供断言失败的错误信息

assert expression [, arguments]

等价于

if not expression:

    raise AssertionError(arguments)

例如：

assert x > 0, "x must be greater than 0!"

当x<=0时，抛出 AssertionError，并附带错误消息 "x must be greater than 0!"。

## Python文件操作

文件操作是一个非常重要的基础概念，它允许你对文件进行读写、修改、删除等操作。Python 提供了丰富的文件操作接口，使得操作文件非常简便。

### 打开文件

* Python open() 方法用于打开一个文件，并返回文件对象。
* 在对文件进行处理过程都需要使用到这个函数，如果该文件无法被打开，会抛出 OSError。
* 注意：

1. 使用 open() 方法一定要保证关闭文件对象，即调用 close() 方法。
2. open() 函数常用形式是接收两个参数：文件名(file)和模式(mode)。

完整语法格式：

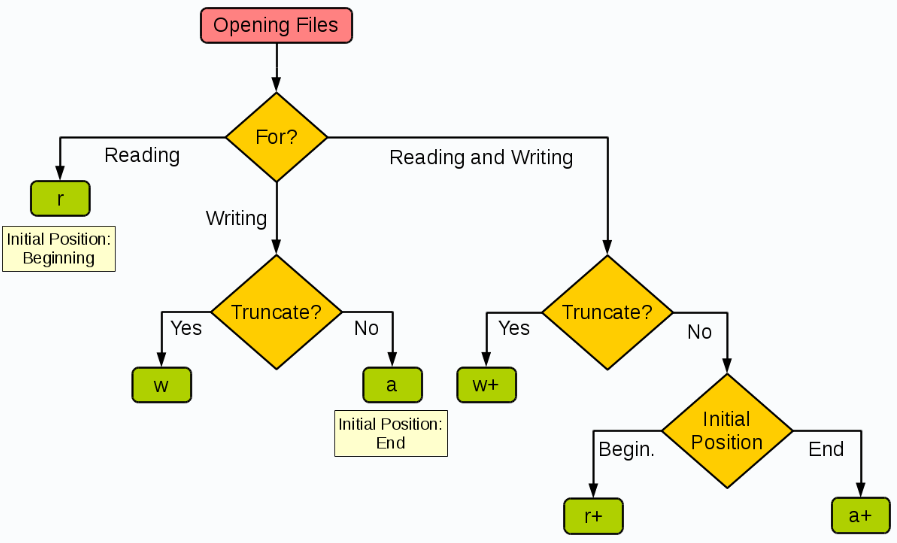
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True,opener=None)

参数说明:

* file: 必需，文件路径（相对或者绝对路径）。
* mode: 可选，文件打开模式
* buffering: 设置缓冲
* encoding: 一般使用utf8
* errors: 报错级别
* newline: 区分换行符
* closefd: 传入的file参数类型
* opener: 设置自定义开启器，开启器的返回值必须是一个打开的文件描述符。

mode参数有：

|  |  |
| --- | --- |
| t | 文本模式 (默认)。 |
| x | 写模式，新建一个文件，如果该文件已存在则会报错。 |
| b | 二进制模式。 |
| + | 打开一个文件进行更新(可读可写)。 |
| r | 以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式 |
| rb | 以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。一般用于非文 本文件如图片等。 |
| r+ | 打开一个文件用于读写。文件指针将会放在文件的开头。 |
| rb+ | 以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。一般用于非文本文件如图片 等。 |
| w | 打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删 除。如果该文件不存在，创建新文件 |
| wb | 以二进制格式打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有 内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等。 |
| w+ | 打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。 如果该文件不存在，创建新文件。 |
| wb+ | 以二进制格式打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内 容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等。 |
| a | 打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将 会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。 |
| ab | 以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是 说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。 |
| a+ | 打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模 式。如果该文件不存在，创建新文件用于读写。 |
| ab+ | 以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文 件不存在，创建新文件用于读写。 |



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 模式 | r | r+ | w | w+ | a | a+ |
| 读 | √ | √ |  | √ |  | √ |
| 写 |  | √ | √ | √ | √ | √ |
| 创建 |  |  | √ | √ | √ | √ |
| 覆盖 |  |  | √ | √ |  |  |
| 指针在头 | √ | √ | √ | √ |  |  |
| 指针在尾 |  |  |  |  | √ | √ |

* with进行上下文资源管理可以省略close()

### 读写文件

* **读文件：**

1. **read()**：读取整个文件内容，并返回一个字符串。
2. **readline()**：读取文件的一行内容，调用多次可以逐行读取。
3. **readlines()**：读取文件的所有行，并返回一个列表，其中每一项是文件中的一行。

* **写文件：**

1. **write()**：写入字符串到文件。如果文件已存在，默认会覆盖原内容。
2. **writelines()**：写入多个字符串（列表或其他可迭代对象），每个字符串将按顺序写入文件。

### file对象

file对象用open()打开，下面是其它常用函数

|  |  |
| --- | --- |
| file.close()  关闭文件。关闭后文件不能再进行读写操作。 | file.isatty() 如果文件连接到一个终端设备返回 True，否则返回 False。 |
| file.fileno()  返回一个整型的文件描述符(file descriptor FD 整型), 可以用在如os模块的read方法等一些底层操作上。 | file.flush()  刷新文件内部缓冲，直接把内部缓冲区的数据立刻写入文件, 而不是被动的等待输出缓冲区写入。 |
| file.read([size])  从文件读取指定的字节数，如果未给定或为负则读取所有。 | file.readline([size])  读取整行，包括 "\n" 字符。 |
| file.readlines([sizeint])  读取所有行并返回列表，若给定sizeint>0，返回总和大约为sizeint字节的行, 实际读取值可能比 sizeint 较 大, 因为需要填充缓冲区。 | file.truncate([size])  从文件的首行首字符开始截断，截断文件为 size 个字符，无 size 表示从当前位置截断；截断之后后面的 所有字符被删除，其中 windows 系统下的换行代表2个字符大小。 |
| file.seek(offset[, whence])  移动文件读取指针到指定位置 | file.tell()  返回文件当前位置。 |
| file.write(str)  将字符串写入文件，返回的是写入的字符长度。 | file.writelines(sequence) 向文件写入一个序列字符串列表，如果需要换行则要自己加入每行的换行符。 |

## Python进阶

### Python迭代器与生成器

##### 迭代器

迭代器是实现了\_\_iter\_\_()和\_\_next\_\_()方法的对象。迭代器可以用于在一个序列上逐个遍历元素。

* 迭代是 Python 最强大的功能之一，是访问集合元素的一种方式。
* 迭代器是一个可以记住遍历的位置的对象。
* 迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。
* 迭代器有两个基本的方法：**iter()创建** 和 **next()取元素**
* 字符串，列表或元组对象都可用于创建迭代器

list=[1,2,3,4]

it = iter(list)    # 创建迭代器对象 print (next(it))

print (next(it))    #1

print (next(it))    #2

迭代器的特点：

* 惰性计算：迭代器会在需要的时候动态计算下一个元素，不会一次性将所有数据加载到内存中。
* 只能遍历一次：迭代器一旦遍历完毕，无法重置或重新遍历，需要重新创建迭代器对象。

###### 创建迭代器

一个类作为一个迭代器使用需要在类中实现两个方法 \_\_iter\_\_() 与 \_\_next\_\_() 。

创建一个返回数字的迭代器，初始值为 1，逐步递增 1：

class MyNumbers:

    def \_\_iter\_\_(self):

        self.a = 1

        return self

    def \_\_next\_\_(self):

        x = self.a

        self.a += 1

        return x

###### StopIteration

StopIteration 异常用于标识迭代的完成，防止出现无限循环的情况，，在 \_\_next\_\_() 方法中我们可以设置在完成指 定循环次数后触发 StopIteration 异常来结束迭代：

def \_\_next\_\_(self):

    if self.a <= 20:

      x = self.a

      self.a += 1

      return x

    else:

      raise StopIteration

##### 生成器

生成器是使用 **yield** 关键字的函数。生成器是一个特殊类型的迭代器，它会返回一个可迭代的对象，并在每次调用时返回一个值。

* 生成器是一个返回迭代器的函数，只用于迭代操作，简单点理解生成器就是一个迭代器。
* 生成器函数中使用 yield 语句时，函数的执行暂停，并将 yield 后面的表达式作为当前迭代的值返回。
* 每次调用生成器的 next() 方法或使用 for 循环进行迭代时，函数会从上次暂停的地方继续执行，直到再次 遇到 yield 语句。这样，生成器函数可以逐步产生值，而不需要一次性计算并返回所有结果。

def countdown(n):

    while n > 0:

        yield n

        n -= 1

# 创建生成器对象

generator = countdown(5)

# 通过迭代生成器获取值

print(next(generator))  # 输出: 5

print(next(generator))  # 输出: 4

print(next(generator))  # 输出: 3

# 使用 for 循环迭代生成器

for value in generator:

    print(value)  # 输出: 2 1

关键点：

* yield：生成器函数中使用 yield 关键字来返回一个值，并且函数的状态会被“挂起”。下一次调用时，会从上次挂起的位置继续执行。
* 惰性计算：与普通迭代器一样，生成器也是惰性求值的，只在需要时计算下一个值。
* 节省内存：生成器不像普通的迭代器一次性返回所有结果，它可以在内存中处理大量数据，而不会消耗过多内存。

### Python装饰器

装饰器（Decorator）是 Python 中一种非常强大的功能，它允许你在不修改函数本身的情况下，动态地改变或增强函数的行为。装饰器通常用于以下场景：日志、权限校验、缓存、性能测试等。

#### 装饰器基础

* 装饰器本质上是一个函数，它接受一个函数作为输入并返回一个函数。装饰器可以修改被装饰函数的行为，或者在其执行之前或之后添加额外的功能。
* 装饰器的核心是 高阶函数：即接收函数作为参数并返回一个函数的函数。
* 基本用法1.decorator(函数名)；2.@ decorator放在函数定义前

基本结构：

def decorator(func):

    def wrapper():

        print("Before function execution")

        func()  # 调用原函数

        print("After function execution")

    return wrapper

@decorator

def say\_hello():

    print("Hello!")

say\_hello()

#加上@decorator前输出：Hello!

#加上后输出：Before function execution

#           Hello!

#           After function execution

理解：

加上装饰器后，主函数调用say\_hello时检测到装饰器后变成：say\_hello（）=decorator（say\_hello），也就是把say\_hello函数变成了里面的wrapper（）函数。由此达成函数内容的替换更新。

#### 带参数的装饰器（工厂）

装饰器函数也可以接受参数，由此定义装饰器的一些行为，这样得到了一个装饰器工厂，例：

def repeat(n):

    def decorator(func):

        def wrapper(\*args, \*\*kwargs):

            for \_ in range(n):

                result = func(\*args, \*\*kwargs)

            return result

        return wrapper

    return decorator

@repeat(3)

def greet(name):

    print(f"Hello, {name}!")

greet("Alice")

# Output:Hello, Alice!

#        Hello, Alice!

#        Hello, Alice!

#### 装饰器的嵌套

装饰器嵌套时，执行顺序是 **先上后下、由内至外**，即从最接近函数定义的装饰器开始执行。

@decorator1

@decorator2

def func（）

等价于

Func（）=decorator2（decorator1（func））

def decorator1(func):

    def wrapper():

        print("Decorator 1 - Before")

        func()

        print("Decorator 1 - After")

    return wrapper

def decorator2(func):

    def wrapper():

        print("Decorator 2 - Before")

        func()

        print("Decorator 2 - After")

    return wrapper

@decorator1

@decorator2

def say\_hello():

    print("Hello!")

say\_hello()

# Output:Decorator 1 - Before

#       Decorator 2 - Before

#       Hello!

#       Decorator 2 - After

#       Decorator 1 - After

### 上下文管理器

上下文管理器的核心功能是提供一组在进入和退出时需要执行的操作，通常用于在代码块执行前后进行资源的初始化与清理。常见的情况是打开文件、数据库连接、线程锁等。

**上下文管理器的基本工作原理**：

* 在 with 语句块开始时执行资源的初始化操作。
* 在 with 语句块结束时（不管是否有异常发生）执行资源的清理操作。

#### 上下文管理器的实现

上下文管理器通过实现 \_\_enter\_\_() 和 \_\_exit\_\_() 方法来定义它的行为：

* **\_\_enter\_\_(self)**：在进入 with 语句块之前执行，返回一个对象（通常是资源管理对象）。
* **\_\_exit\_\_(self, exc\_type, exc\_value, traceback)**：在 with 语句块执行完毕后（无论是否发生异常）执行，负责清理资源，exc\_type, exc\_value, traceback 提供了异常信息（如果有的话）。

#### with语句

with 语句用于简化上下文管理器的使用。其语法如下：

with expression as variable:

    # 使用资源的代码块

#### 以文件操作为例子

with open('example.txt', 'r') as file:

    content = file.read()

    print(content)

# 不需要显式地调用 file.close()，文件会在块结束时自动关闭

这个例子中：

* open() 返回的文件对象是一个上下文管理器。
* \_\_enter\_\_() 方法打开文件并返回文件对象。
* \_\_exit\_\_() 方法在文件操作完后自动关闭文件。

其实就是使用上下文管理器可以自动关闭文件，因为文件对象中已经实现enter、exit功能。

### 多线程

## 常用库与框架

### PyMySQL

### Json

### Requests

requests 是 Python 中用于发送 HTTP 请求的非常流行的库，它为与 HTTP 服务的交互提供了一个非常简洁和直观的接口。它被广泛用于 Web 开发、API 调用、自动化测试、数据抓取等场景。

知识前备：[1.3.6.5 HTTP协议](#_HTTP协议（HyperText_Transfer_Protocol）)

#### 核心原理

requests 库的核心原理是将底层的 HTTP 请求过程封装为更高层次的 API，使得开发者能够以更加简洁和直观的方式进行 HTTP 通信。它通过内部集成的 http.client、urllib 等库来处理底层的网络请求和响应，同时提供了很多便捷的功能，如会话管理、连接池、异常处理等。

requests 基于几个核心模块进行构建：

* **http.client**：Python 内置的 HTTP 客户端模块，负责与 Web 服务器建立连接并发送 HTTP 请求。
* **urllib**：用于处理 URL 编码、解码、参数拼接等工作。
* **socket**：Python 的低级网络接口，负责底层的网络通信。
* **certifi**：一个外部库，提供最新的 CA 证书集，以确保 SSL 连接的安全性。
* **chardet** 或 **cchardet**：用于检测和解码响应内容的字符编码。

#### 请求方法

requests 支持常见的 HTTP 请求方法：

* **GET**：获取资源

response = requests.get('https://api.example.com/resource')

* **POST**：提交数据

response = requests.post('https://api.example.com/resource', data={'key': 'value'})

* **PUT**：更新资源

response = requests.put('https://api.example.com/resource', data={'key': 'new\_value'})

或者直接request动态使用

requests.request(method, url, params=params, json=data)

|  |  |
| --- | --- |
| **方法** | **描述** |
| delete(*url*, *args*) | 发送 DELETE 请求到指定 url |
| get(*url*, *params, args*) | 发送 GET 请求到指定 url |
| head(*url*, *args*) | 发送 HEAD 请求到指定 url |
| patch(*url*, *data, args*) | 发送 PATCH 请求到指定 url |
| post(*url*, *data, json, args*) | 发送 POST 请求到指定 url |
| put(*url*, *data, args*) | 发送 PUT 请求到指定 url |
| request(*method*, *url*, *args*) | 向指定的 url 发送指定的请求方法 |

#### 请求参数

使用请求方法时，有时候须有附带一些API请求参数，可以通过 URL 参数（params）、请求头（headers）、请求体（data 或 json）向服务器发送数据。

##### URL参数（GET）

通过设置params可以设置get请求时的URL参数

可以通过**字典（默认、常用）、列表（多值参数）、字节（不自动编码）**进行参数传递

import requests

params = {

    "page": 1,

    "sort": "desc",

    "filter": "python"

}

response = requests.get(

    "https://api.example.com/search",

    params=params  # 自动编码为 ?page=1&sort=desc&filter=python

)

###### 基本特性

* **自动编码**

requests 自动处理

* ​**​URL 编码​**​：将特殊字符（如空格、&、=）转换为 %20、%26、%3D 等。
* ​**​类型转换​**​：字典的值可以是字符串、整数、列表等，requests 会将其转换为字符串。
* ​**​参数拼接​**​：将键值对拼接为 key1=value1&key2=value2 的格式。

params = {"q": "Python & APIs"}

# 编码后：?q=Python%20%26%20APIs

上面例子就是自动把“ & ”转化成“%20%26%20”，然后把q（键）和Python & APIs（值）进行参数拼接

* **多值参数**

若值为 list 或 tuple，生成多个同名参数

params = {"tags": ["web", "dev"]}

# 生成: ?tags=web&tags=dev

* **空值忽略**

None 会被忽略，不添加到 URL 中

params = {"query": "python", "details": None}

# 生成: ?query=python

###### 特殊情况

* **与现有 URL 参数合并​​**

若 URL 本身已有参数，params 会与其合并（重复参数名会被覆盖）。

url = "https://api.example.com/data?page=1"

params = {"page": 2, "sort": "desc"}

# 最终 URL: https://api.example.com/data?page=2&sort=desc

* **​保留特殊字符​​**

若需保留某些字符（如 / 或 :）不被编码，需手动拼接 URL：

url = "https://api.example.com/data?path=unencoded/path"

* **​嵌套参数（需序列化）​​**

若需要传递嵌套结构（如 JSON），需手动序列化为字符串：

import json

nested\_params = {"filters": {"language": "python", "year": 2023}}

params = {"q": json.dumps(nested\_params)}

# 生成: ?q=%7B%22language%22%3A%20%22python%22%2C%20%22year%22%3A%202023%7D

##### 请求头

* 设置请求头（Headers）是通过 headers 参数实现的。请求头是 HTTP 请求的重要组成部分，用于传递客户端的元数据（如浏览器信息、内容类型、认证令牌等）。
* 常用于设置UA、Host、Cookie等参数。
* 在发送请求时，通过 headers 参数传入一个字典，键为请求头字段名，值为对应的值。也可以在请求方法中一个个参数设置

headers = {

    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/119.0.0.0 Safari/537.36",

    "Accept": "application/json",

    "Authorization": "Bearer YOUR\_ACCESS\_TOKEN",

}

response = requests.get(

    "https://api.example.com/data",

    headers=headers

)

###### 常见请求头字段

|  |  |  |
| --- | --- | --- |
| **字段名** | **作用** | **示例值** |
| User-Agent | 标识客户端类型（如浏览器或爬虫） | "python-requests/2.31.0"（默认值） |
| Content-Type | 请求体的数据类型 | "application/json" |
| Accept | 声明客户端可接收的响应类型 | "text/html" |
| Authorization | 认证信息（如 Token 或 Basic Auth） | "Bearer YOUR\_TOKEN" |
| Cookie | 传递 Cookie | "sessionid=abc123; username=john" |
| Referer | 标识请求来源的页面 URL | "https://www.example.com/login" |
| X-Requested-With | 标识请求类型（如 AJAX） | "XMLHttpRequest" |

###### 注意事项

* **键的大小写不敏感​​**

请求头字段名不区分大小写（如 user-agent 和 User-Agent 等效），但建议统一使用首字母大写的格式。

* ​​**避免无效字段​​**

不要添加无意义的请求头字段，某些服务器可能会拒绝包含非标准头的请求。

* ​​**敏感信息处理​​**

避免在代码中硬编码敏感信息（如 API 密钥），应通过环境变量或配置文件读取：

import os

headers = {"Authorization": f"Bearer {os.getenv('API\_TOKEN')}"}

* ​​**响应头获取​​**

服务器返回的响应头可通过 response.headers 访问

###### 特殊技巧

* **覆盖默认请求头​​**

requests 库默认会添加以下请求头：

User-Agent: python-requests/2.31.0

Accept-Encoding: gzip, deflate

Accept: \*/\*

Connection: keep-alive

通过自定义 headers 参数可以覆盖这些默认值。

* **动态生成请求头​​**

如果某些头字段需要动态生成（如时间戳或签名），可以在字典中动态赋值：

import time

headers = {

    "User-Agent": "MyCustomClient/1.0",

    "X-Timestamp": str(int(time.time())),

    "X-Signature": generate\_signature(),  # 自定义签名生成函数

}

* **复用请求头（使用 Session 对象）​​**

对于需要多次请求的场景，使用 Session 对象可以全局复用 headers，避免重复设置：

with requests.Session() as session:

    session.headers.update({

        "Authorization": "Bearer YOUR\_TOKEN",

        "Accept": "application/json",

    })

    # 所有通过 session 发送的请求都会自动携带这些 headers

    response1 = session.get("https://api.example.com/data1")

    response2 = session.get("https://api.example.com/data2")

* **处理 JSON 请求​​**

当发送 JSON 数据时，需明确设置 Content-Type 为 application/json：

headers = {"Content-Type": "application/json"}

data = {"key": "value"}

response = requests.post(

    "https://api.example.com/submit",

    headers=headers,

    json=data  # 自动将 data 序列化为 JSON 字符串

)

* **防止反爬虫​​**

模拟浏览器请求头，避免被网站封禁：

headers = {

    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/119.0.0.0 Safari/537.36",

    "Accept-Language": "en-US,en;q=0.9",

    "Referer": "https://www.google.com/",

}

##### 请求体

请求体（Request Body）用于在 ​**​POST、PUT、PATCH​**​ 等非 GET 请求中向服务器发送数据。

通过以下参数处理请求体：

|  |  |  |
| --- | --- | --- |
| **参数** | **作用** | **适用场景** |
| data | 发送表单数据、文本或二进制内容（自动编码或手动处理） | 表单提交、原始文本、字节流 |
| json | 自动将 Python 对象序列化为 JSON 字符串，并设置 Content-Type 头 | REST API 的 JSON 请求 |
| files | 上传文件或二进制数据（支持多文件上传） | 文件上传、多部分表单 |

###### 发送表单数据（data 参数）​​

适用于 application/x-www-form-urlencoded 格式的表单提交。

import requests

payload = {

    "username": "user123",

    "password": "secret",

}

response = requests.post(

    "https://api.example.com/login",

    data=payload,  # 自动编码为 username=user123&password=secret

    headers={"User-Agent": "MyApp/1.0"}

)

###### 发送 JSON 数据（json 参数）​​

自动序列化字典为 JSON 字符串，并设置 Content-Type: application/json。

import requests

payload = {

    "name": "Python",

    "version": "3.11",

    "tags": ["programming", "language"]

}

response = requests.post(

    "https://api.example.com/create",

    json=payload,  # 自动序列化为 JSON 字符串

)

也可以通过文件读取本地json文件传入

###### 上传文件（files 参数）​​

支持单文件、多文件上传，或自定义文件名。

import requests

# 单文件上传（自动设置多部分编码）

with open("report.pdf", "rb") as f:

    files = {"file": f}

    response = requests.post("https://api.example.com/upload", files=files)

# 多文件上传（同一字段名）

files = [

    ("images", ("photo1.jpg", open("photo1.jpg", "rb"))),

    ("images", ("photo2.jpg", open("photo2.jpg", "rb"))),

]

response = requests.post(url, files=files)

# 自定义文件名和 Content-Type

files = {

    "document": ("report.pdf", open("report.pdf", "rb"), "application/pdf")

}

response = requests.post(url, files=files)

###### 其它

* ​​发送原始文本或二进制数据（data 参数），直接传递字符串或字节流，需手动设置 Content-Type。
* 多部分混合请求（表单 + 文件）​​，结合 data 和 files 参数发送混合数据。

###### 注意事项

* ​​**编码问题​​：**

使用 data 发送字符串时，默认编码为 utf-8。如需其他编码，可手动转换

* ​​**性能优化​​：**

大文件上传使用 files 参数会自动启用流式传输，避免内存占用过高。

* ​​**调试请求体​​：**

打印已编码的请求体（需在发送前捕获）：

import requests

from requests\_toolbelt.utils import dump

response = requests.post(url, data=payload)

print(dump.dump\_all(response).decode("utf-8"))

* ​​**Content-Type 自动设置​​：**
* files 参数会自动设置 multipart/form-data。
* json 参数会自动设置 application/json。
* 手动使用 data 时需指定 Content-Type。

###### 特殊情况

* **嵌套 JSON 或复杂结构​​**

使用 json 参数直接序列化嵌套对象：

* ​​**流式上传大文件​​**

避免内存溢出，分块读取文件：

with open("large\_file.zip", "rb") as f:

    # 分块读取（每次 1MB）

    for chunk in iter(lambda: f.read(1024 \* 1024), b""):

        requests.post(url, data=chunk, headers={"Content-Type": "application/octet-stream"})

* ​​**禁用自动重定向或处理响应​​**

**重定向（Redirection）**是网络通信中的一种机制，指​**​客户端（如浏览器或程序）在请求某个资源时，服务器返回一个指示，要求客户端自动转向另一个 URL 或路径​**​。

通过 allow\_redirects 和流式响应控制行为：

import requests

response = requests.post(

    url,

    data=payload,

    allow\_redirects=False,  # 禁止自动重定向

    stream=True            # 流式接收响应

)

#### 响应内容

response = requests.get()

在requests 库中，requests.get()、post()、put() 等请求方法返回的是一个 ​**​requests.models.Response​**​ 对象。这个对象包含了服务器返回的全部响应信息，可以通过其属性和方法提取数据、状态码、头信息等。

##### Respose对象

* **属性**

|  |  |
| --- | --- |
| **属性** | **说明** |
| **​​status\_code​​** | **HTTP 状态码（如 200 成功，404 未找到，500 服务器错误）。** |
| **​​headers​​** | **响应头的字典（键不区分大小写），包含服务器元数据（如 Content-Type）。** |
| **​​url​​** | **实际请求的最终 URL（自动处理重定向后的地址）。** |
| **​​history​​** | **所有重定向的 Response 对象列表（按请求顺序）。** |
| **​​cookies​​** | **服务器返回的 RequestsCookieJar 对象，包含所有 Cookie。** |
| **​​text​​** | **响应内容的字符串形式（自动根据 encoding 解码）。** |
| **​​content​​** | **响应内容的二进制字节形式（适用于图片、文件等非文本数据）。** |
| **​​encoding​​** | **响应内容的字符编码（如 utf-8，可手动修改）。** |
| **​​elapsed​​** | **请求耗时（datetime.timedelta 对象，如 0:00:01.234567）。** |
| **​​request​​** | **关联的 PreparedRequest 对象（包含实际发送的请求头、方法、URL 等）。** |
| **​​raw​​** | **底层的 urllib3 原始响应对象（需 stream=True）。** |
| **​​reason​​** | **HTTP 状态码的文本描述（如 200 → OK）。** |
| **​​is\_redirect​​** | **布尔值，表示响应是否是重定向（状态码为 3xx）。** |
| **​​is\_permanent\_redirect​​** | **布尔值，表示是否是永久重定向（301、308）。** |
| **​​links​​** | **解析响应头中的 Link 字段，返回链接关系字典（常用于分页或资源关联）。** |
| **​​ok​​** | **布尔值，快速判断 status\_code 是否小于 400（即请求是否成功）。** |

* **方法**

|  |  |
| --- | --- |
| **方法** | **说明** |
| **​​json(​\*\*​kwargs)​​** | **解析 JSON 响应体为 Python 对象，解析失败抛出 JSONDecodeError。获取响应体** |
| **​​raise\_for\_status()​​** | **若状态码为 4xx 或 5xx，抛出 HTTPError 异常。** |
| **​​iter\_content(chunk\_size=1)​​** | **生成器，按指定块大小（字节）流式读取二进制内容（用于大文件下载）。** |
| **​​iter\_lines(chunk\_size=512)​​** | **生成器，按行流式读取文本内容（需 stream=True）。** |
| **​​close()​​** | **手动关闭响应连接（通常由上下文管理器自动处理）。** |
| **​​next​​** | **获取重定向链中下一个请求的 PreparedRequest 对象（需 allow\_redirects=False）。** |

* **Header属性**

# 获取特定头字段（键不区分大小写）

content\_type = response.headers.get("Content-Type")

server = response.headers["Server"]

|  |  |
| --- | --- |
| **字段名** | **典型值示例** |
| **Content-Type** | **text/html; charset=utf-8** |
| **Content-Length** | **1024** |
| **Location** | **https://new-url.com（重定向地址）** |
| **Set-Cookie** | **session\_id=abc123; Path=/** |
| **Cache-Control** | **max-age=3600** |
| **Server** | **nginx/1.18.0** |

##### 内容提取

通过访问Response对象可以提取响应内容

#### Session与持久化配置

​**​Cookie​**​ 和 ​**​Session​**​ 是处理 HTTP 会话状态和身份验证的核心机制

##### Cookie

* **获取 Cookie​​**：从响应对象中提取。

import requests

response = requests.get("https://example.com/login")

cookies = response.cookies  # 获取 Cookie 对象（RequestsCookieJar）

# 提取单个 Cookie

session\_id = cookies.get("session\_id")

* ​​**发送 Cookie​​：**通过 cookies 参数添加到请求。

cookies = {"session\_id": "abc123"}

response = requests.get("https://example.com/profile", cookies=cookies)

##### Session

​​Session​​ 是 requests 提供的会话对象，用于在多个请求之间 ​​自动保留 Cookie、请求头、认证信息等​​，模拟浏览器行为。

* 自动处理 Cookie 的存储和传递。
* 重用 TCP 连接，提升性能（连接池机制）。
* **创建 Session​​：**

session = requests.Session()

* ​​**使用 Session 发送请求​​：**

# 登录（自动保存 Cookie）

login\_data = {"username": "user", "password": "pass"}

session.post("https://example.com/login", data=login\_data)

# 后续请求自动携带 Cookie

response = session.get("https://example.com/profile")

* ​​**全局设置请求头/认证​​：**

session.headers.update({"User-Agent": "MyApp/1.0"})

session.auth = ("user", "pass")

### Selenium/Playwright



* **Selenium** 和 **Playwright** 都是用于自动化浏览器操作的工具，主要用于Web应用程序的自动化测试和爬虫开发。它们有一些相似之处，但也有各自的特点。
* **Selenium** 更为稳定，适合传统的自动化需求，而 **Playwright** 则在新兴的Web自动化和性能方面更具优势，适合需要更高效率和更复杂交互的场景。

下面简单介绍Selenium与Playwright，详细见8.4.web自动化

#### Selenium

##### Selenium 特点

* 跨浏览器支持：Selenium 支持多种主流浏览器，如 Chrome、Firefox、Safari、Internet Explorer 等，可以在不同浏览器中执行相同的测试脚本。
* 多语言支持：Selenium 支持多种编程语言，开发者可以用自己熟悉的语言来编写测试脚本。
* 灵活性：Selenium 可以与各种测试框架结合使用，如 TestNG、JUnit、PyTest 等，提供了丰富的测试功能。
* 与 CI/CD 集成：Selenium 可以与 Jenkins 等持续集成工具集成，帮助实现自动化测试和持续交付。
* 自动化Web交互：Selenium 可以模拟真实用户的操作，如点击、滚动、键盘输入等

##### Selenium组成

Selenium 由多个组件组成，每个组件都有其特定的用途。以下是 Selenium 的主要组件：

* **Selenium WebDriver**：这是 Selenium 的**核心组件**，用于直接与浏览器进行交互。WebDriver 提供了丰富的 API，允许开发者通过代码控制浏览器的行为，如打开网页、点击按钮、填写表单等。
* **Selenium IDE**：这是一个浏览器插件，主要用于录制和回放用户的操作。Selenium IDE 适合初学者快速创建简单的测试脚本，但它不支持复杂的逻辑和条件判断。
* **Selenium Grid**：这是一个用于并行执行测试的工具。通过 Selenium Grid，你可以在多个浏览器和操作系统上同时运行测试，从而加快测试速度并提高测试覆盖率。

##### Selenium WebDriver

Selenium WebDriver 是 Selenium 的核心组件，它提供了与浏览器交互的 API，允许开发者通过编程语言控制浏览器并执行各种操作。

* WebDriver需要自行根据浏览器类型下载。
* 通过webdriver的初始化来配置文件下载、UA、有无头等配置。

##### Selenium 八大定位法

1. 通过 ID 定位

方法：find\_element(By.ID, "id\_value")

说明：通过元素的 id 属性定位。

from selenium.webdriver.common.by import By

element = driver.find\_element(By.ID, "username")

1. 通过 Name 定位

方法：find\_element(By.NAME, "name\_value")

说明：通过元素的 name 属性定位。

element = driver.find\_element(By.NAME, "password")

1. 通过 Class Name 定位

方法：find\_element(By.CLASS\_NAME, "class\_name")

说明：通过元素的 class 属性定位。

element = driver.find\_element(By.CLASS\_NAME, "submit-btn")

1. 通过 Tag Name 定位

方法：find\_element(By.TAG\_NAME, "tag\_name")

说明：通过元素的标签名定位（如 <div>、<input> 等）。

element = driver.find\_element(By.TAG\_NAME, "input")

1. 通过 CSS 选择器定位

方法：find\_element(By.CSS\_SELECTOR, "css\_selector")

说明：通过 CSS 选择器定位元素。

element = driver.find\_element(By.CSS\_SELECTOR, "input#username")

1. 通过 XPath 定位

方法：find\_element(By.XPATH, "xpath\_expression")

说明：通过 XPath 表达式定位元素。

element = driver.find\_element(By.XPATH, "//input[@id='username']")

1. 通过 Link Text定位

方法：find\_element(By.LINK\_TEXT, "link\_text")

说明：通过链接的文本内容定位（适用于 <a> 标签）。

element = driver.find\_element(By.LINK\_TEXT, "Click Here")

1. 通过 Partial Link Text 定位

方法：find\_element(By.PARTIAL\_LINK\_TEXT, "partial\_link\_text")

说明：通过链接的部分文本内容定位。

element = driver.find\_element(By.PARTIAL\_LINK\_TEXT, "Click")

##### Selenium 元素操作

|  |  |  |
| --- | --- | --- |
| **输入文本** | send\_keys("text") | 向输入框或文本区域输入指定的文本。字符串 |
| **点击元素** | click() | 点击按钮、链接或其他可点击的元素。 |
| **清除输入框内容** | clear() | 清除输入框或文本区域中的内容。 |
| **获取元素属性** | get\_attribute("attribute\_name") | 获取元素的指定属性值（如 value、href、class 等） |
| **获取元素文本** | text | 获取元素的可见文本内容 |
| **复选框/单选框操作** | is\_selected() 和 click() | 检查复选框或单选框是否被选中，并选中或取消选中。 |
| **下拉列表操作** | **Select 类**的 select\_by\_index、select\_by\_value、select\_by\_visible\_text | 通过索引、值或可见文本选择下拉列表的选项。 |
| **鼠标操作** | **ActionChains 类**的 click、double\_click、context\_click、drag\_and\_drop、move\_to\_element | 执行复杂的鼠标操作，如点击、双击、拖放、悬停等。 |
| 键盘操作 | **Keys 类**的 send\_keys(Keys.KEY\_NAME) | 模拟键盘操作，如按下回车键、Tab 键、组合键等 |
| **文件上传** | send\_keys("file\_path") | 通过 <input type="file"> 元素上传文件。 |
| **执行 JavaScript** | execute\_script("javascript\_code") | 执行 JavaScript 代码，用于复杂的操作或修改页面内容。 |

##### Selenium 等待机制

**隐式等待**

* 隐式等待是一种全局性的等待机制，通过 implicitly\_wait() 方法来设置，它会在查找元素时等待一定的时间。如果在指定的时间内找到了元素，Selenium 会立即继续执行后续操作；如果超时仍未找到元素，则会抛出 NoSuchElementException 异常。
* 隐式等待的优缺点
* 优点：
* 简单易用，只需设置一次即可应用于所有元素查找操作。
* 适用于大多数简单的场景。
* 缺点：
* 全局性等待，可能会导致不必要的等待时间。
* 无法处理某些复杂的等待条件，例如等待元素变为可点击状态。

**显式等待**

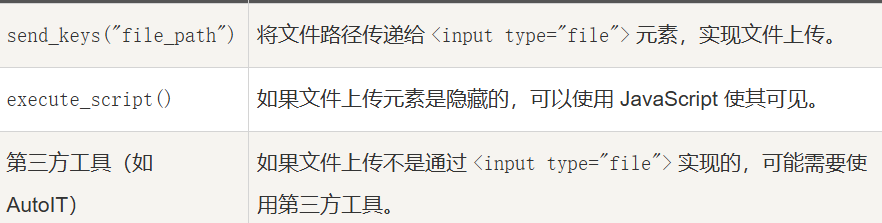
* 显式等待是一种更为灵活的等待机制，它允许你为特定的操作设置等待条件。显式等待通常与 WebDriverWait 类和 expected\_conditions 模块一起使用。可以设置timeout、frequency、ignored等参数。
* expected\_conditions 模块提供了多种预定义的等待条件，以下是一些常用的条件：
* presence\_of\_element\_located：等待元素出现在 DOM 中。
* visibility\_of\_element\_located：等待元素出现在 DOM 中并且可见。
* element\_to\_be\_clickable：等待元素可点击。
* text\_to\_be\_present\_in\_element：等待元素的文本包含指定的文本。
* 显式等待的优缺点
* 优点：
  + 灵活性高，可以为不同的操作设置不同的等待条件。
  + 可以处理复杂的等待场景。
* 缺点：
* 代码相对复杂，需要更多的代码量。
* 需要为每个操作单独设置等待条件。

**固定等待**

固定等待是一种最简单的等待机制，它通过 time.sleep() 方法让脚本暂停执行指定的时间。无论页面是否加载完成，脚本都会等待指定的时间后再继续执行。

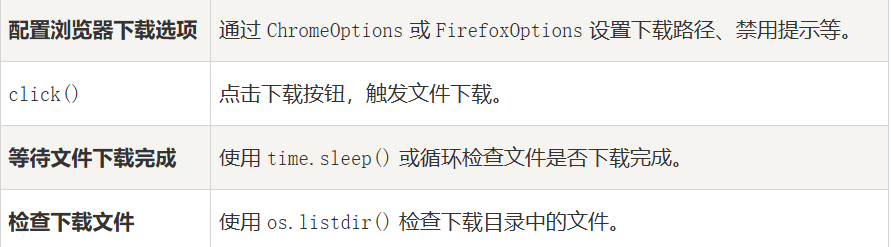
##### Selenium 文件上传与下载

* **文件上传**



* **文件下载**

由于selenium无法操作系统窗口，所以文件下载往往需要设置静默下载



##### Selenium 测试框架

软件测试框架的封装概念指的是将常用的测试操作、逻辑和配置封装成可复用的模块或类，以提高测试代码的可维护性、可扩展性和效率。

* **主要内容：**

1. 封装测试逻辑：将页面操作、元素定位、验证逻辑等封装成独立的函数或类，避免重复代码。这样，测试用例只需关注具体的测试场景，而不必处理细节。
2. 模块化：测试框架将不同功能的代码分为不同的模块（如页面对象、测试用例、测试数据、日志管理等），各模块之间解耦，便于维护和扩展。
3. 可复用性：通过封装，能够将通用的测试逻辑复用到多个测试用例中，提高测试效率，减少重复劳动。
4. 易于扩展：随着项目需求的变化，可以方便地对框架进行扩展和修改，比如添加新的测试功能或更改已有的逻辑，而不影响其他部分。

* **封装的好处：**

1. 提高代码复用性，减少重复。
2. 增强测试脚本的可读性和可维护性。
3. 易于调试和扩展，尤其在大型项目中非常有效。

* **简单示例：**

在页面对象模型（Page Object Model）中，封装了页面的操作逻辑，如登录操作、按钮点击等。测试用例只需调用这些封装好的方法进行验证。

#### Playwright

Playwright 是由 Microsoft 开发的一款开源自动化测试工具，主要用于现代 Web 应用的端到端（E2E）测试、浏览器自动化以及网页爬虫开发。它支持 ​**Chrome**、**Firefox**、**Safari** 和 ​**Edge** 等主流浏览器，并提供跨平台（Windows/macOS/Linux）和多语言（JavaScript/TypeScript、Python、Java、.NET）的支持。

##### playwright的分层管理

* Playwright 通过三个层次（Browser、BrowserContext、Page）完成资源隔离和职责分离，实现了高效的浏览器自动化管理。不同于Selenium初始化一个webdriver即可，playwright通过分层次初始化层级元素来实现更灵活的资源管理。
* browser是最高层，类比电脑的物理浏览器程序，用于管理浏览器进程的启动于关闭。browser下是context层，类比用户的开启的浏览器隐身模式会话窗口，但用户可以开启多个浏览器窗口，且这些串口相互隔离。context下是page层，类似一个浏览器窗口开启多个页面标签。
* 不同层次负责管理不同的资源或行为

|  |  |  |  |
| --- | --- | --- | --- |
| **功能** | **Browser 层** | **Context 层** | **Page 层** |
| ​**进程管理** | ✅ 启动/关闭 | ❌ | ❌ |
| ​**会话隔离** | ❌ | ✅ Cookie/Storage | ❌ |
| ​**页面导航** | ❌ | ❌ | ✅ goto() |
| ​**网络拦截** | ❌ | ✅ 全局拦截 | ✅ 页面级拦截 |
| ​**设备模拟** | ❌ | ✅ 设备/UA/时区 | ❌ |
| ​**文件下载** | ❌ | ✅ 设置下载目录 | ✅ 监听下载事件 |
| ​**视频录制** | ❌ | ✅ 录制整个 Context | ❌ |
| ​**权限控制** | ❌ | ✅ 地理位置/通知 | ❌ |

##### playwright的同步与异步

在 Playwright 的 Python API 中，有两种方式来使用其功能：同步 和 异步。

**核心区别**：同步适合快速开发，异步专注极致性能；选择时需权衡开发效率与执行效率，多数场景同步足够，密集 I/O 操作优先异步。

|  |  |  |
| --- | --- | --- |
| **特性** | **同步模式** | **异步模式** |
| ​**语法** | 无 async/await | 需使用 async/await |
| ​**执行模型** | 单线程阻塞 | 基于事件循环的非阻塞 |
| ​**性能** | 适合简单任务 | 高并发场景性能更优 |
| ​**适用场景** | 简单脚本、线性操作 | 并行任务、复杂异步逻辑 |
| ​**兼容性** | 兼容所有 Python 环境 | 需 Python 3.7+ 和 asyncio 支持 |

* **同步模式**

同步 API 是基于传统的阻塞式编程模型。在这种模型下，程序执行会按照代码的顺序逐行进行，每一行代码都需要等待前一行执行完成才能继续执行。

* 特点：

1. 代码简单易懂，符合大多数开发者的编程习惯。
2. 所有操作按顺序执行，适合不涉及高并发、长时间等待的场景。
3. 执行效率较低，特别是当有多个独立任务需要同时进行时（如等待多个网络请求时），代码可能会被阻塞。

* 使用方式： Playwright 提供了一个同步的 API 版本，方便直接像普通的同步代码一样使用。

from playwright.sync\_api import sync\_playwright

with sync\_playwright() as p:

    browser = p.chromium.launch()

    page = browser.new\_page()

    page.goto('https://example.com')

    print(page.title())

* **异步模式**

异步 API 基于 Python 的 asyncio 库，允许在执行一个操作时不会阻塞其他操作，这使得代码能够同时执行多个任务，提升性能和效率，尤其是在需要等待外部资源（如页面加载、请求响应等）时。

* 特点：
  1. 非阻塞：可以在等待某些操作时继续执行其他任务，提高效率，尤其在处理多个并发任务时（如同时加载多个页面或发送多个请求）。
  2. 需要异步编程知识：使用 async 和 await 关键字，要求开发者理解并正确使用异步编程模型。
  3. 执行效率高，适用于需要处理多个并发请求、页面加载等场景。
* 使用方式： Playwright 提供了异步的 API，可以使用 asyncio 来运行异步代码。

import asyncio

from playwright.async\_api import async\_playwright

async def run():

    async with async\_playwright() as p:

        browser = await p.chromium.launch()

        page = await browser.new\_page()

        await page.goto('https://example.com')

        print(await page.title())

        await browser.close()

asyncio.run(run())

##### playwright浏览器管理

|  |  |  |
| --- | --- | --- |
| **方法** | **功能说明** | **示例** |
| launch() | 启动浏览器实例 | browser = p.chromium.launch(headless=False) |
| new\_context() | 创建独立会话上下文 | context = browser.new\_context(user\_agent='自定义UA') |
| close() | 关闭浏览器及其所有页面 | browser.close() |

使用with上下文管理器可以自动进行资源管理

##### playwright的页面操作、元素获取、元素交互

* **页面操作**

|  |  |  |
| --- | --- | --- |
| **方法** | **功能说明** | **典型应用场景** |
| goto(url) | 导航到指定URL | page.goto("https://example.com") |
| screenshot() | 页面/元素截图 | page.screenshot(path="screenshot.png") |
| evaluate() | 执行JavaScript代码 | title = page.evaluate("document.title") |
| wait\_for\_timeout() | 强制等待 | page.wait\_for\_timeout(3000) |
| reload() | 重新加载页面 | page.reload() |
| bring\_to\_front() | 切换标签页焦点 | page.bring\_to\_front() |

* **元素获取**

|  |  |  |
| --- | --- | --- |
| **方法** | **功能说明** | **返回值类型** |
| **selector（）** | **默认获取第一个元素** | **ElementHandle** |
| query\_selector() | 获取单个元素 | ElementHandle |
| query\_selector\_all() | 获取多个元素 | List[ElementHandle] |
| wait\_for\_selector() | 等待元素出现 | ElementHandle |
| is\_visible() | 检查元素可见性 | Boolean |
| get\_attribute() | 获取元素属性 | String |

* **元素交互**

|  |  |  |
| --- | --- | --- |
| **方法** | **功能说明** | **元素定位示例** |
| click(selector) | 点击元素 | page.click("button#submit") |
| fill(selector, text) | 填充输入框 | page.fill("input[name='user']", "admin") |
| check(selector) | 勾选复选框/单选框 | page.check("#agree\_terms") |
| select\_option(selector, value) | 选择下拉选项 | page.select\_option("#country", "CN") |
| hover(selector) | 鼠标悬停 | page.hover(".menu-item") |
| type(selector, text) | 模拟键盘输入 | page.type("#search", "Playwright") |

##### playwright的文件上传与下载

* **文件上传**

Playwright 提供了 locator.set\_input\_files() 方法来直接将文件路径传递给文件输入框，从而模拟文件上传。

* **文件下载**

Playwright不需要主动设置静默下载，其默认使用（accept\_downloads=True），也不需要主动设置文件保存地址，只需要通过监听下载事件管理文件下载。

with page.expect\_download() as download\_info:

page.locator(macro\_xpaths.download\_btn).click()   #下载文件

download = download\_info.value

download.save\_as(os.path.join(dl\_dir, download.suggested\_filename))

### pytest

pytest 是一个功能强大的 Python 测试框架，设计简洁且易于使用，广泛用于单元测试、集成测试和功能测试。它支持自动化测试用例的发现和执行，提供了强大的断言功能、丰富的插件支持和灵活的配置选项，能够简化测试过程和提高开发效率。pytest 的核心特点包括支持简单的测试函数、可扩展的插件体系、灵活的测试夹具（fixtures）管理以及自动生成测试报告，是 Python 开发者进行自动化测试的首选工具之一。

详细介绍可见[Pytest框架](#_Pytest框架)