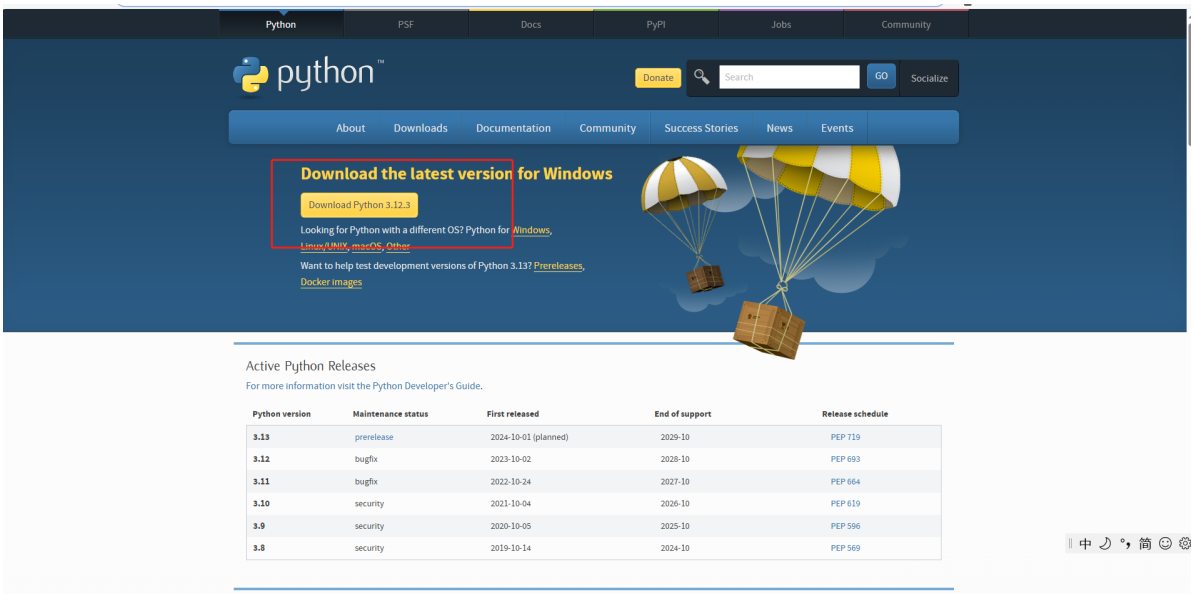
python基础学习

python的安装

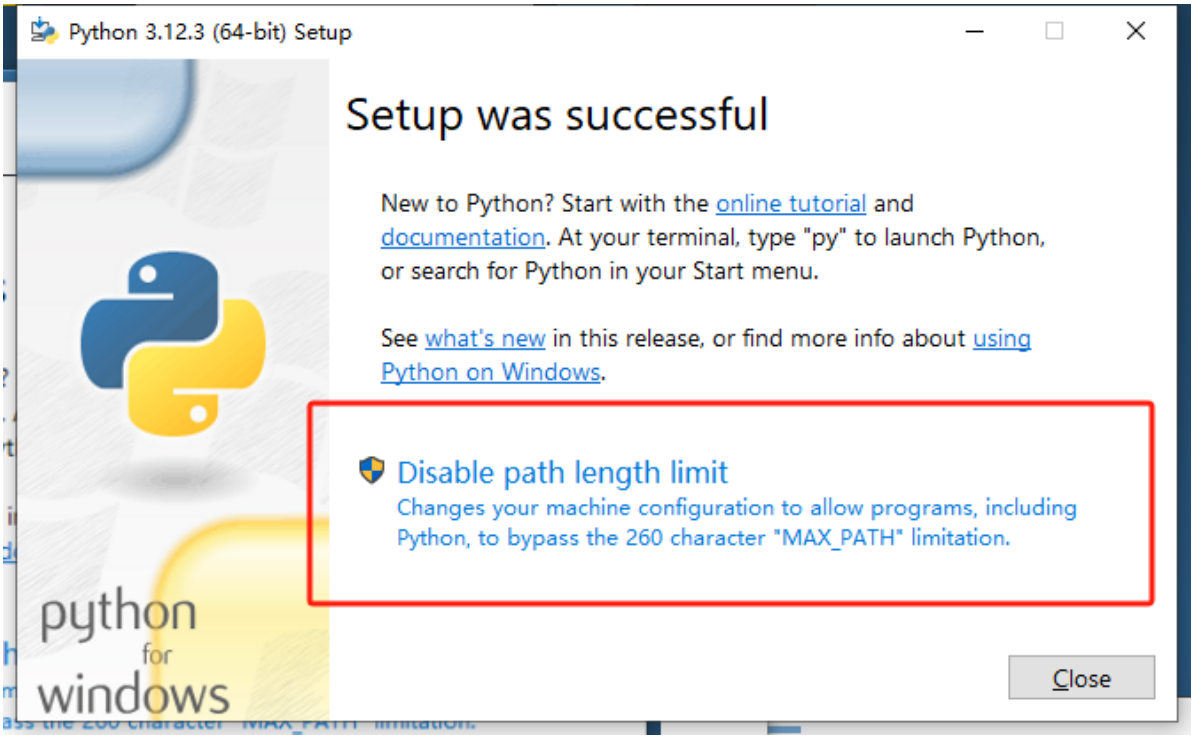
1、进入python官网下载python[Download Python | Python.org](https://www.python.org)



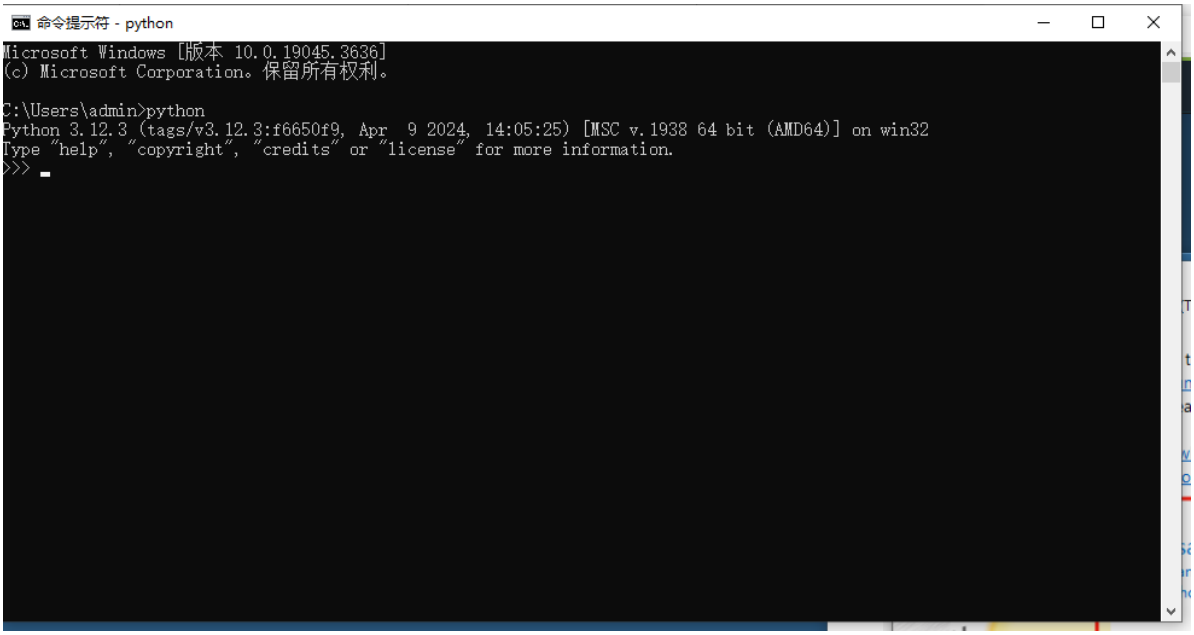
2、下载完成后直接点击自定义安装（注意勾选下方的自动创建路径）

3、选择安装路径，点击安装

4、勾选disable path length limit

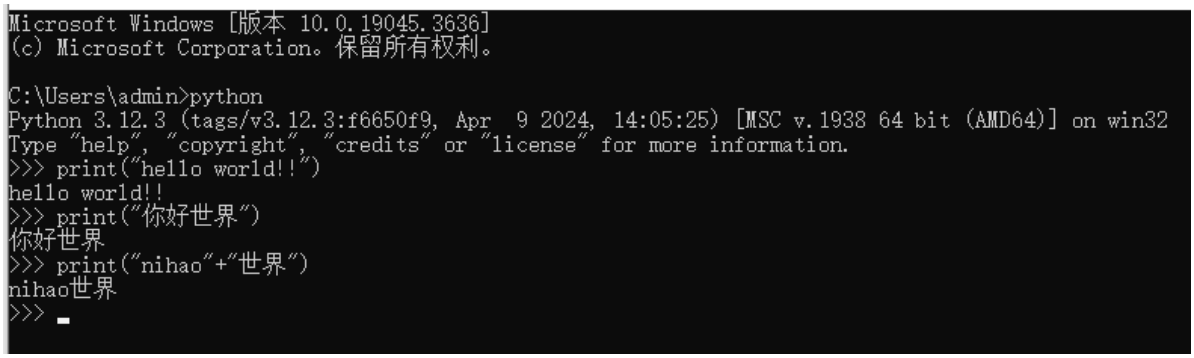


5、安装完成，控制台输入python检查是否安装成功



第一个python程序

打开cmd并输入python，出现三个“>”后可以直接输入python程序



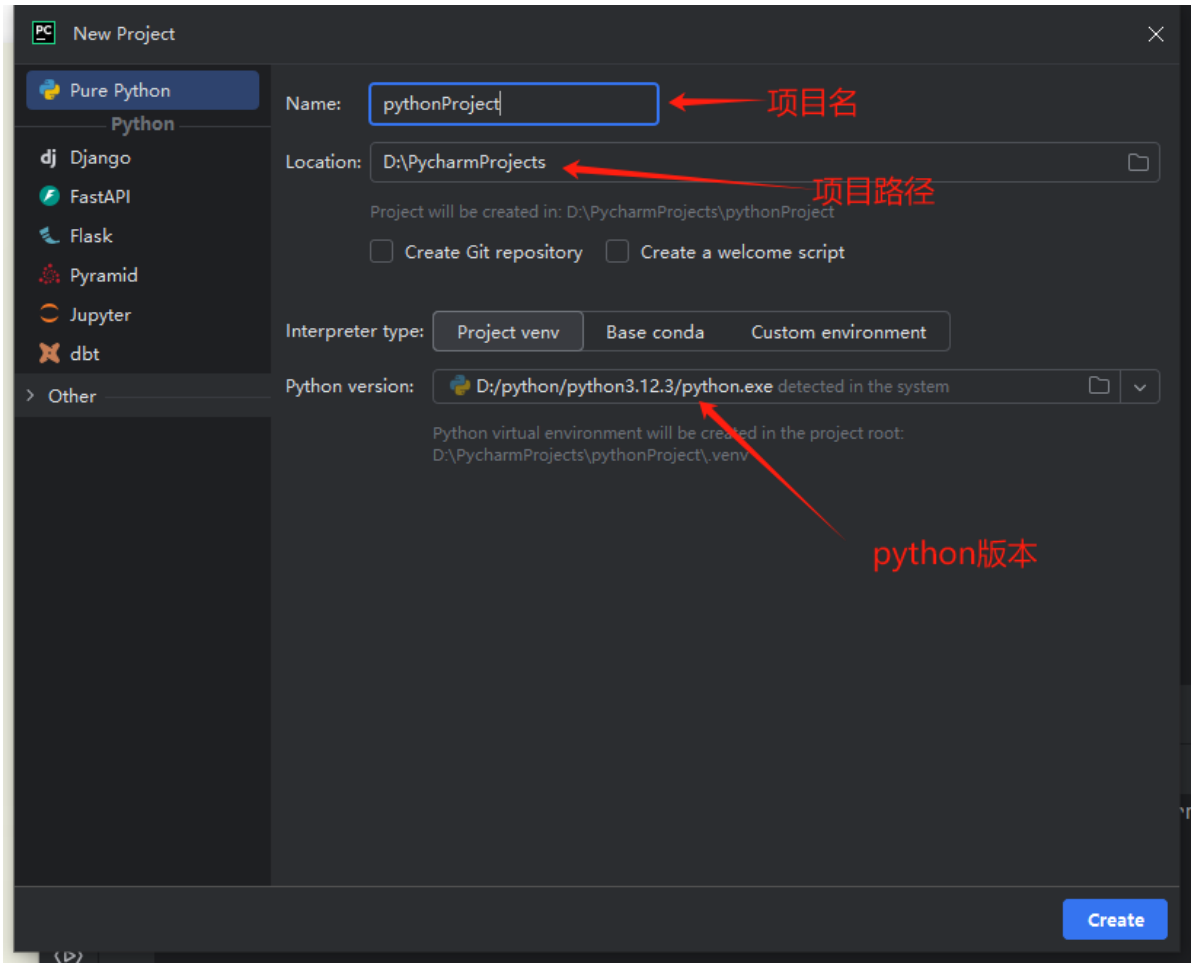
安装PyCharm

官网直接安装pycharm https://www.jetbrains.com/

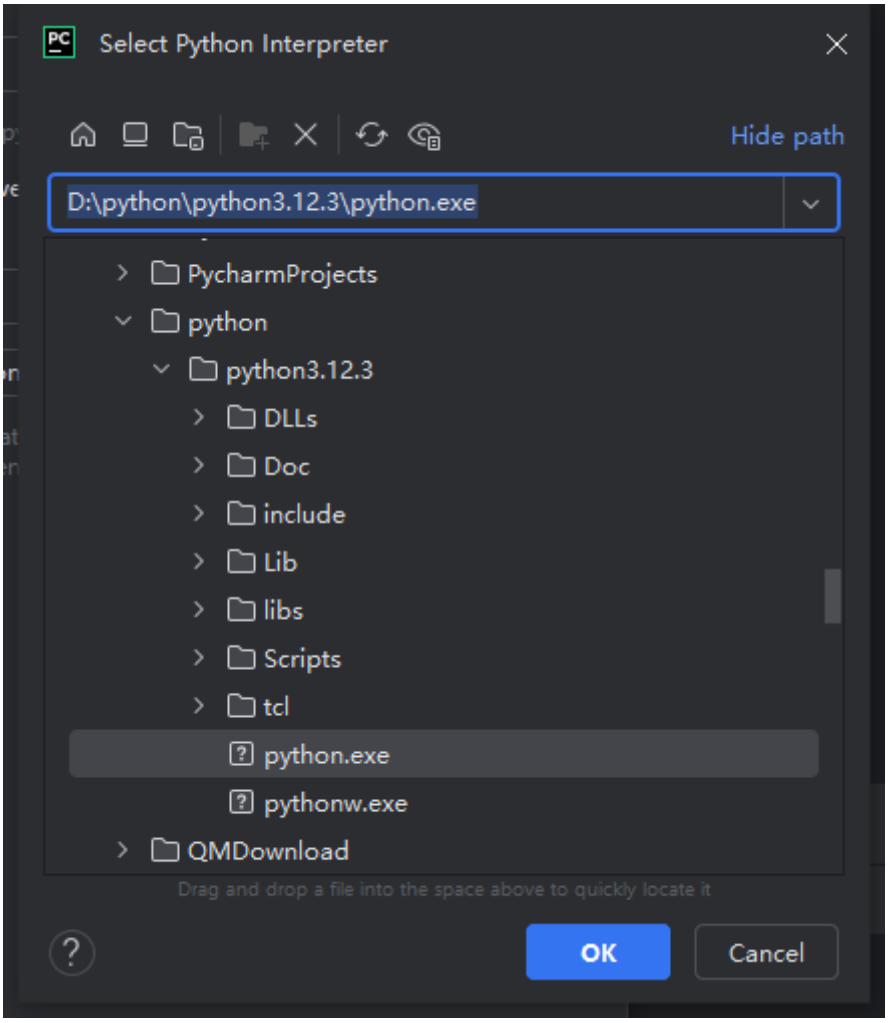
创建第一个python项目

1、打开pycharm，并选择创建项目

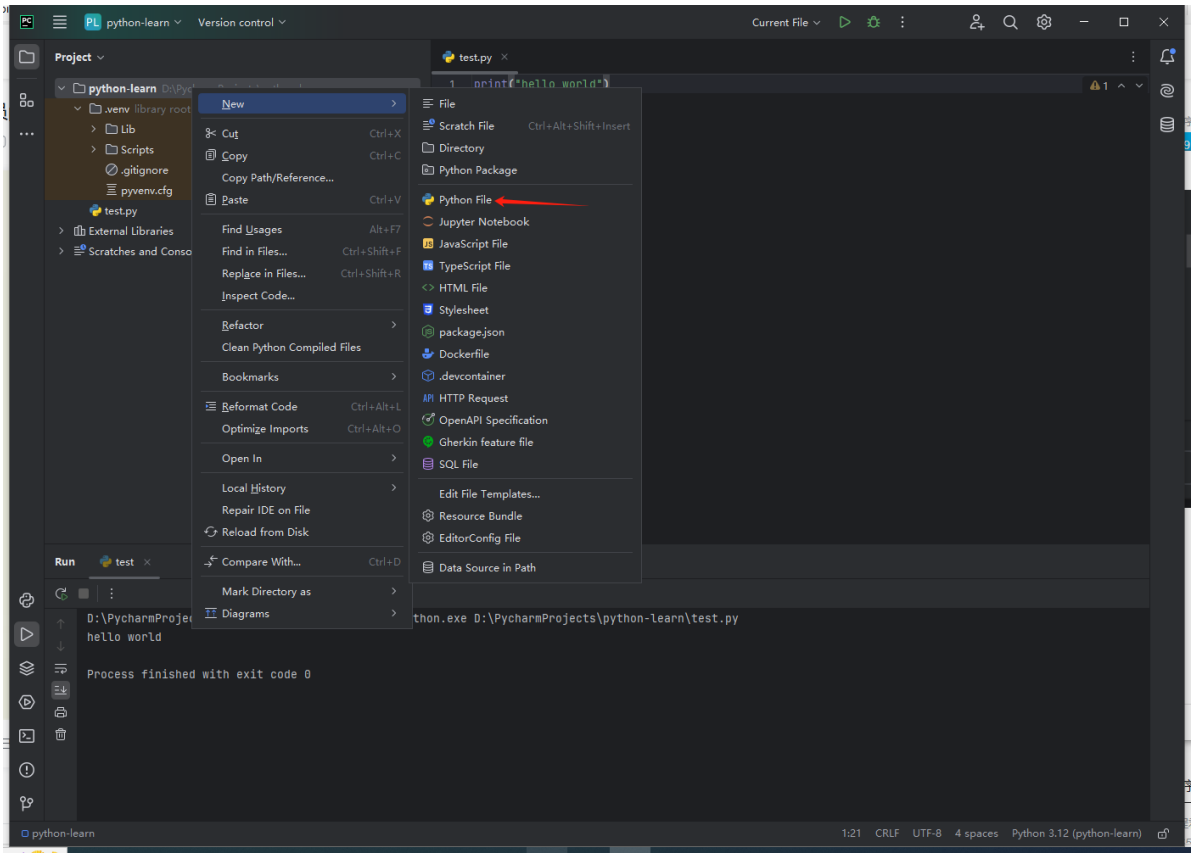
2、输入项目名和python版本



注：如果python版本没有指定对，需要进入python安装目录设置对应的python.exe路径



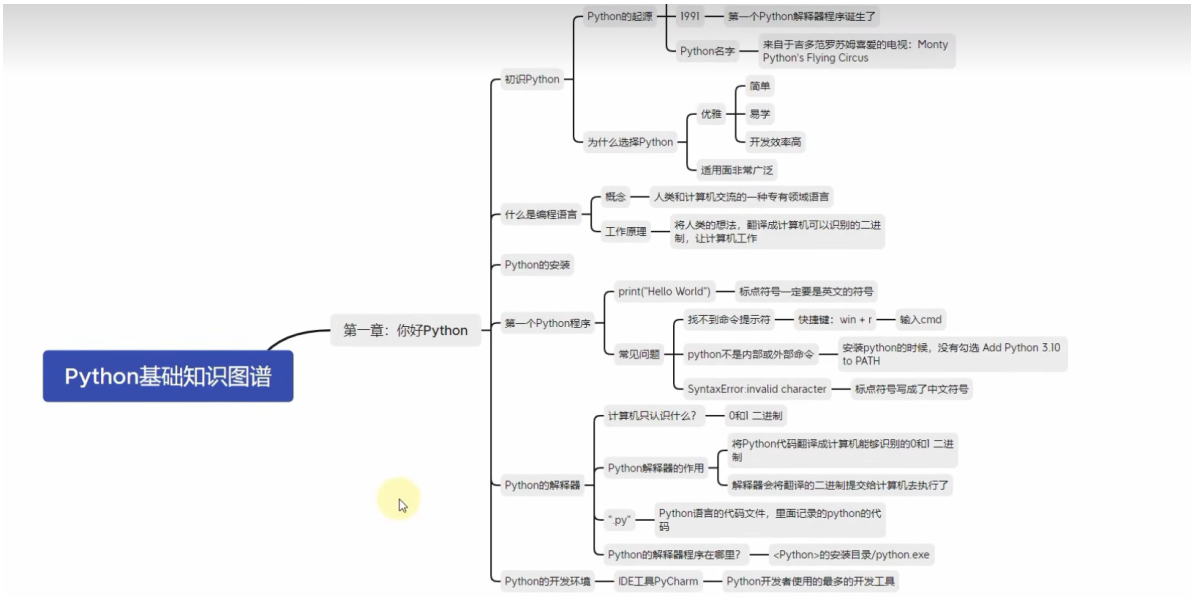
3、点击创建，并右键选择创建python file



4、编写程序

5、pycharm常用快捷键

```
ctrl+alt+s:打开软件设置
ctrl+d:复制当前行代码
shift+alt+上\下:将当前行代码上移或下移
crtl+shift+f10:运行当前代码文件
shift+f6:重命名文件
ctrl+a:全选
ctrl+c\v\x:复制、粘贴、剪切
ctrl+f:搜索
```



Python基本语法

1、字面量

什么是字面量?

字面量：在代码中，被写下来的固定的值，称之为字面量

常用的值类型

Python中常用的有6种值（数据）的类型

类型	描述	说明
数字（Number）	支持 • 整数（int） • 浮点数（float） • 复数（complex） • 布尔（bool）	整数（int），如：10、-10
		浮点数（float），如：13.14、-13.14
		复数（complex），如：4+3j，以j结尾表示复数
		布尔（bool）表达现实生活中的逻辑，即真和假，True表示真，False表示假。 True本质上是一个数字记作1，False记作0
字符串（String）	描述文本的一种数据类型	字符串（string）由任意数量的字符组成
列表（List）	有序的可变序列	Python中使用最频繁的数据类型，可有序记录一堆数据
元组（Tuple）	有序的不可变序列	可有序记录一堆不可变的Python数据集合
集合（Set）	无序不重复集合	可无序记录一堆不重复的Python数据集合
字典（Dictionary）	无序Key-Value集合	可无序记录一堆Key-Value型的Python数据集合

2、注释

```
单行注释：以'#'开头
多行注释：以一队三个双引号引起来 """多行注释"""
```

3、变量

变量：在程序运行时，能存储计算结果或能够表示值的抽象概念，简单来说，变量就是在程序运行时用来记录数据用的

```
"""
变量
"""

# 记录钱余额
money = 50
# 输出钱包余额
print("钱包余额还剩：", money)

# 花费10元
money -= 10
# 输出钱包剩余余额
print("钱包余额还剩：", money)
```

4、数据类型

```
money = 100
number = 11.11
name = "zhangsan"
print(type(money))
print(type(number))
my_tyoe = type(name)
print(my_tyoe)
```

5、数据类型转换

- 数据类型转换，将会是我们以后经常使用的功能，

如:

- 从文件中读取的数字，默认是字符串，我们需要转换成数字类型
- 后续学习的input()语句，默认结果是字符串，若需要数字也需要转换
- 将数字转换成字符串用以写出到外部系统
- 等等

常见的转换语句

语句(函数)	说明
int(x)	将x转换为一个整数
float(x)	将x转换为一个浮点数
str(x)	将对象 x 转换为字符串

```
money = 112.11
# 将float转换为int类型
print(int(money))

number = 12
# 将int类型转换为float类型
print(float(number))
# 将int类型转换为string类型
str_number = str(number)
# 输出str_number类型
print(type(str_number))

print("number" + str_number)
# 将str类型转换为float类型
float_number = float(str_number)

# 输出float_number的类型
print("float_number type:" + str(type(float_number)))
```

输出：

```
D:\PycharmProjects\python-learn\.venv\Scripts\python.exe D:\PycharmProjects\python-learn\语法\类型转换.py
112
12.0
<class 'str'>
number12
float_number type:<class 'float'>

Process finished with exit code 0
```

6、标识符

什么是标识符

在Python程序中，我们可以给很多东西起名字，比如:

- 变量的名字
- 方法的名字
- 类的名字，等等

这些名字，我们把它统一的称之为标识符，用来做内容的标识。

所以，标识符:

是用户在编程的时候所使用的一系列名字，用于给变量、类、方法等命名。

命名规则：

- 内容限定
- 大小写敏感
- 不可使用关键字

标识符仅限于英文，中文，数字，下划线，数字不可以用在开头

运算符

运算符	描述	实例
+	加	两个对象相加 a + b 输出结果 30
-	减	得到负数或是一个数减去另一个数 a - b 输出结果 -10
*	乘	两个数相乘或是返回一个被重复若干次的字符串 a * b 输出结果 200
/	除	b / a 输出结果 2
//	取整除	返回商的整数部分 9//2 输出结果 4 , 9.0//2.0 输出结果 4.0
%	取余	返回除法的余数 b % a 输出结果 0
**	指数	a**b 为10的20次方， 输出结果 100000000000000000000

除法会自动转换为float类型

运算符	描述	实例
+=	加法赋值运算符	c += a等效于c = c + a
-=	减法赋值运算符	c -= a等效于c = c - a
*=	乘法赋值运算符	c *= a等效于c = c * a
/=	除法赋值运算符	c /= a等效于c = c / a
%=	取模赋值运算符	c %= a等效于c = c % a
**=	幂赋值运算符	c **= a等效于c = c ** a
//=	取整除赋值运算符	c //= a等效于c = c // a
=	赋值运算符	

7、字符串

字符串的三种定义方式

- 单引号定义法：name = 'youxin'
- 双引号定义法：name = "youxin"
- 三引号定义法：name = """youxin"""

三引号定义法，和多行注释的写法一样，同样支持换行操作

使用变量接收它，它就是字符串

不使用变量接收它，就可以作为多行注释使用

问：如果想要定义的字符串本身就是包含引号自身呢？

- 单引号定义法，可以内含双引号
- 双引号定义法，可以内含单引号
- 可以使用转移字符(\)来将引号接触效用，变成普通字符串

```
# 单引号
name = 'youxin'
name_4 = '"youxin"'
print(name, type(name))
print(name_4, type(name_4))

# 双引号
name_1 = "youxin"
name_5 = "'youxin'"
print(name_1, type(name_1))
print(name_5, type(name_5))

# 三引号
name_2 = """youxin"""
print(name_2, type(name_2))

# 转移字符
name_6 = "\"youxin\""
print(name_6, type(name_6))
```

输出：

```
D:\PycharmProjects\python-learn\.venv\Scripts\python.exe D:\PycharmProjects\python-learn\语法\字符串.py
youxin <class 'str'>
"youxin" <class 'str'>
youxin <class 'str'>
'youxin' <class 'str'>
youxin <class 'str'>
"youxin" <class 'str'>
```

字符串的拼接方式

如果有两个字符串，可以直接用`+`进行拼接

字符串的格式化

可以通过以下语法完成字符串的拼接

```
age = "22"
message = "youxin age is %s" % age
print(message)
```

其中的%s

- `%`表示：我要占位
- `s`表示：将变量变成字符串放入占位的地方

```python
# 单个占位符
age = "22"
message = "youxin age is %s" % age
print(message)

# 多个占位符,同时包含数字和字符串
num =11111
msg = "youxin age is %s and num is %s" % (age, num)
print(msg)
```

格式符号	转化
%s	将内容转换成字符串，放入占位位置
%d	将内容转换成整数，放入占位位置
%f	将内容转换成浮点型，放入占位位置

字符串格式化的精度控制

可以使用辅助符号"m.n"来控制数据的宽度和精度

- `m`：控制宽度，要求是数字（很少使用），设置的宽度小于数字自身，不生效，当数字整数宽度小于设置的宽度时，会自动用空格进行填充
- `.n`：控制小数点精度，要求是数字，会进行小数的四舍五入

字符串格式化方式2

通过语法：`f"内容{变量}"` 的格式来快速格式化，但对float类型不作精度控制

```python
name_7 = "youxin"
age = 22
num = 11111.11
print(f"he name is {name_7},age is {age}, num is {num}") # 不做精度控制
```

输出：

```
he name isyouxin,age is 22, num is 11111.11
```

对表达式格式化

```python
# 对表达式进行格式化
print("1 + 1 = %d" % (1 + 1))
print(f"1 + 1 = {1 + 1}")
print("\'python\'字符串的类型是：%s" % type("python"))
```

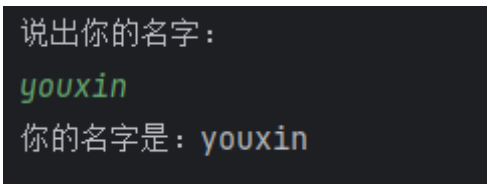
```
1 + 1 = 2
1 + 1 = 2
'python'字符串的类型是：<class 'str'>
```

format()函数

- 主要格式：print('{}{}'.format('字符串1','字符串2'))
- foramt会把参数按位置顺序来填充到字符串中（按照顺序自动分配），第一个参数是0，然后1 .....
- 也可以不输入数字，则会按照顺序自动分配，而且一个参数可以多次插入
- 一个参数可以多次插入
- {{}}仅代表{}，不占用字符串格式化位置顺序

数据输入（input语句）

```
print("说出你的名字：")
name = input()
print("你的名字是：%s" % name)
# input语句可以载要求使用者输入内容前，输出图示内容
age = input("说出你的年龄：")
print(f"你的年龄是：{age}岁")
```



判断

布尔类型的定义

布尔类型的字面量：

- `True` 表示真（是、肯定）
- `False` 表示假（否、否定）

比较运算符

运算符	描述	示例
==	判断内容是否相等，满足为True，不满足为False	如a=3,b=3，则(a == b) 为 True
!=	判断内容是否不相等，满足为True，不满足为False	如a=1,b=3，则(a != b) 为 True
>	判断运算符左侧内容是否大于右侧 满足为True，不满足为False	如a=7,b=3，则(a > b) 为 True
<	判断运算符左侧内容是否小于右侧 满足为True，不满足为False	如a=3,b=7，则(a < b) 为 True
>=	判断运算符左侧内容是否大于等于右侧 满足为True，不满足为False	如a=3,b=3，则(a >= b) 为 True
<=	判断运算符左侧内容是否小于等于右侧 满足为True，不满足为False	如a=3,b=3，则(a <= b) 为 True

判断

格式：

```
if 要判断的条件：    ###冒号不能丢失
    条件成立时，要做的事情 ### 首行需要缩进4个字符
```

判断是否在一个if下：条件成立时要做的事情都需要首行缩进4个字符，如果不首行缩进，则表示与上一个if平级

```
num = 1
if num > 10:
    print("大于十")
    print("大大大")
print("111")
```

if else用法

```
if 条件:
    满足条件时要做的事情1
    满足条件时要做的事情2
    满足条件时要做的事情3
    ...
else:
    不满足条件时要做的事情1
    不满足条件时要做的事情2
    不满足条件时要做的事情3
    ...
```

```
num = 1
if num > 10:
    print("大于十")
else:
    print("小于十")
print("111")
```

if elif else用法

```
if 条件1:
    条件1满足应做的事情
    条件1满足应做的事情
    ...
elif 条件2:
    条件2满足应做的事情
    条件2满足应做的事情
    ...
elif 条件N:
    条件N满足应做的事情
    条件N满足应做的事情
    ...
else:
    所有条件都不满足应做的事情
    所有条件都不满足应做的事情
    ...
```

```python
# 示例
grade = int(input("输入成绩："))
if grade >= 90:
    print("优秀")
elif (grade >= 80) & (grade < 90):
    print("良好")
elif (grade >= 70) & (grade < 80):
    print("及格")
else:
    print("不及格")
```

判断语句的嵌套

其基本语法格式如下：

```
if 条件1:
    满足条件1做的事情1
    满足条件1做的事情2
    if 条件2:
        满足条件2做的事情1
        满足条件2做的事情2
```

嵌套的关键：**空格缩进**来体现层次关系

while循环

基本语法格式如下：

```
while 条件:
    条件满足时，做的事情1
    条件满足时，做的事情2
    条件满足时，做的事情3
    ......
```

注：

1、while的条件需得到布尔类型，True表示继续循环，False表示结束循环

2、需要设置循环终止的条件，如 i += 1 配合 i < 100 ，就能确保100次后停止，否则将无限循环

3、空格缩进和if判断一样，都需要设置

while循环的嵌套和if判断的嵌套一样，要注意空格缩进

for循环

for循环和while循环区别：

- while循环的循环条件时自定义的，自行控制循环条件
- for循环是一种“轮询”机制，是对一批内容进行“逐个处理“
- for循环是无法定义循环条件的
- 只能从被处理的数据集中，依次取出内容进行处理，所以理论上讲，python的for循环无法构建无限循环（被处理的数据集不可能无限大）

for循环遍历字符串

```
name = "youxin"
for i in name:
    print(i)
```

for循环语法

```
for 临时变量 in 待处理的数据集
    循环满足条件时执行的代码
```

语法中的：待处理的数据集，严格来说，称之为：序列类型

序列类型指：其内容可以一个个依次取出的一种类型，包括：

- 字符串
- 列表
- 元祖
- 等

range语句

语法1：`range(num)`，获取一个从0开始，到num结束的数字序列（不含num本身）

如`range(5)`取得的数据是：[0, 1, 2, 3, 4]

语法2：`range(num1, num2)`，取得一个从num1开始，到num2结束的数字序列（不含num2本身）

如`range(5, 10)`取得的数据是[5, 6, 7, 8, 9]

语法3：`range(num1, num2, step)`取得一个从num1开始，到num2结束的数字序列（不含num2本身），数字之间的步长，以step为准（step默认为1）

如`range(5, 10, 2)`取得的数据是[5, 7, 9]

```
for i in range(5):
    print(i)

for i in range(5, 10):
    print(i)

for i in range(5, 10, 2):
    print(i)
```

break和continue

break：退出整个循环

continue：退出当前循环，继续开始下一次循环

python函数

函数：是组织好的，可重复使用的，用来实现特定功能的代码段。

函数的定义：

```
def 函数名(传入参数):
    函数体
    return 返回值
```

注：

- 参数如不需要，可以省略
- 返回值如不需要，可以省略
- 函数必须先定义后使用
- 参数可以没有，也可以有任意多个，但形参和实参个数必须相同

函数返回值

None类型

思考：如果函数没有使用return语句返回数据，那么函数有返回值吗？

实际上是有的。Python中有一个特殊的字面量：None，其类型是：<class 'NoneType'>无返回值的函数，实际上就是返回了`None`这个字面量

```
def none():
    print()


result = none()
print(result)
print(type(result))
```

```
None
<class 'NoneType'>
```

注：可以显示返回None类型，其结果和不设置返回数据一样

None类型的应用场景

- 用在函数无返回值上
- 用在if判断上：

  在if判断中，None等同于False

  一般用于在函数中主动返回None，配合if判断做相关处理

```
def check_age(age):
    if age > 18:
        return True
    return None


result_age = check_age(9)
if result_age:
    print("已成年")
else:
    print("未成年")
```

- 用于声明无内容的变量上

  定义变量，但暂时不需要变量有具体值，可以用None来代替

```
name = None
```

函数的说明文档

函数式纯代码语言，想要理解其含义，就需要一行行的去阅读理解代码，效率比较低，这是可以给函数添加说明文档，辅助理解函数的作用，其语法如下：

```
def func(x, y):
    """
    函数说明
    :param x: 形参x的说明
    :param y: 形参y的说明
    """
    函数体
    return 返回值
```

在pycharm中使用三个引号再回车，会自动生成文档模板

变量在函数中的作用域

全局变量：在函数内部和外部均可使用

局部变量：作用范围在函数内部，在函数外部无法使用

> 如果在函数内容想要修改全局变量，默认会将函数内的变量设置为局部变量，从而无法在函数内部修改全局变量

global关键字

使用`global`关键字，可以在函数内部声明变量为全局变量，如：

```
num = 100
def test():
    # 声明mun是全局变量
    global num
    num = 200
    print(num)

test()
```

函数多返回值

如果一个函数有多个返回值，按照返回值的顺序，写对应顺序的多个变量接收即可，变量之间用逗号隔开，且支持不同类型的数据return

```
def test_return():
    return 1,2

x, y = test_return()
print(x)    # 结果1
print(y)    # 结果2
```

函数的多种参数使用形式

函数参数种类

使用方式上的不同，函数有4中常见参数使用方式：

- 位置参数：调用函数时根据函数定义的参数位置来传递参数

```
def user_info(name, age):
    print(name, age)

user_info("zhangsan", 24)
```

传递的参数和定义的参数的顺序及个数必须一致

- 关键字参数：函数调用时通过"键 = 值"形式传递参数

```
def user_info(name, age):
    print(name, age)

user_info(age = 24, name = "zhangsan")
user_info("lisi", age = 22)
```

关键字参数可以和位置参数混用，但位置参数必须放在最前面

- 缺省参数：缺省参数也叫默认参数， 用于定义函数函数，为参数提供默认值，调用函数时可不传该默认参数的值(注意:所有位置参数必须出现在默认参数前，包括函数定义和调用)

```
def user_info(name, age = 22):
    print(name, age)

user_info("zhangsan",  25)
user_info("lisi")
```

函数调用时，如果为缺省参数传值则修改默认参数值，否则使用这个默认值

- 不定长参数：不定长参数也叫可变参数，用于不确定调用的时候会传递多少个参数（不传参也可以）的场景。不定长参数的类型：1、位置传递，2、关键字传递

```
# 位置传递
def user_info(*args):
    print(args)

user_info("tom")
user_info("tom", 24)
```

传进的所有参数都会被args变量收集，它会根据传进参数的位置合并为一个元组(tuple)，args是元组类型，这就是位置传递。

```
# 关键字传递
def user_info(**kwargs):
    print(kwargs)

# {"name": "tom", "age": 21, "id": 111}
user_info(name = "tom", age = 21, id = 111)
```

参数是“键=值”形式的形式的情况下，所有的“键=值”都会被kwargs接受，同时会根据“键=值”组成字典。

匿名函数

函数作为参数传递

如下代码：

```
def test_func(compute):
    result = compute(1, 2)
    print(result)

def compute(x, y):
    return x + y

test_func(compute)
```

函数compute，作为参数，传入了test func函数中使用

- test_func需要一个函数作为参数传入，这个函数需要接收2个数字进行计算，计算逻辑由这个被传入函数决定
- compute函数接收2个数字对其进行计算，compute函数作为参数，传递给了test_func函数使用
- 最终，在test_func函数内部，由传入的compute函数，完成了对数字的计算操作

所以，这是一种计算逻辑的传递，而非数据的传递，任何逻辑都可以自行定义并作为函数传入

lambda匿名函数

函数的定义中：

- def关键字，可以定义带有名称的函数
- lambda关键字，可以定义匿名函数（无名称）

有名称的函数，可以基于名称重复使用，无名称的匿名函数，只可以临时使用一次。

匿名函数定义语法：

`lambda 传入参数: 函数体(一行代码)`

- lambda是关键字，表示定义匿名函数
- 传入参数表示匿名函数的形式参数，如：x, y表示接收两个形式参数
- 函数体，就是函数的执行逻辑，要注意：只能写一行，无法写多行代码

```
def test_func(compute):
    result = compute(1, 2)
    print(result)

test_func(lambda x, y: x +y) # 不用写return语句，默认自动return
```

数据容器

数据容器是什么？一种可以容纳多份数据的数据类型，容纳的每一份数据称之为1个元素，每一个元素可以是任意类型的数据，如字符串、数字、布尔等。

数据容器根据特点的不同，如：

- 是否支持重复元素
- 是否可以修改

- 是否有序，等

分为5类，分别是：

列表（list）、元组（tuple）、字符串（str）、集合（set）、字典（dict）

数据容器：list列表

基本语法：

```
# 字面量
[元素1, 元素2, 元素3, ...]

# 定义变量
变量名称 = [元素1, 元素2, 元素3, ...]

# 定义空列表
变量名称 = []
变量名称 = list()
```

注：列表可以一次存储多个数据，且可以为不同的数据类型，支持嵌套

列表的下标（索引）

如何从列表中取出特定位置的数据：可以使用下标索引（通过下标从0开始，依次递增）

或者，可以使用反向索引，就是从后往前，从-1开始，依次递减（-1，-2，-3...）

而如果是嵌套列表，可以通过外层加内层取出。如list[0][1]取出

列表的查询功能（方法）

- 查找某元素的下标

功能：查找指定元素在列表的下标，如果找不到，报错ValueError

语法：`列表.index(元素)`

- 修改特定位置（索引）的元素值

语法：`列表[下标] = 值`

可以使用如上语法，直接对指定下标的值进行修改

- 插入元素

语法：`列表.insert(下标, 元素)`，在指定的下标位置，插入指定的元素

- 追加元素

语法：`列表.append(元素)`，将指定的元素，追加到列表尾部

- 追加元素方式2

语法：`列表.extend(其他数据容器)`，将其他数据容器的内容取出，依次追加到列表尾部

- 删除元素

语法1：`del 列表[下标]`

语法2：`列表.pop(下标)`，不但可以删除下标元素，同时可以使用一个参数进行接收所删除的下标元素

- 删除某元素在列表中的第一个匹配项

语法：`列表.remove(元素)`

- 清空列表内容

语法：`列表.clear()`

- 统计某元素在列表内的数量

语法：`列表.count(元素)`

- 统计容器内有多少元素

语法：`len(列表)`

列表的特点

- 可以容纳多个元素(上限为2**63-1、9223372036854775807个)
- 可以容纳不同类型的元素(混装)
- 数据是有序存储的(有下标序号)
- 允许重复数据存在
- 可以修改(增加或删除元素等)

```
# 定义列表并用变量接收
mylist = [21, 25, 21, 23, 22, 20]

# 追加一个数字31到列表的尾部
mylist.append(31)
print(mylist)

# 追加一个新列表[29, 33, 30]到列表的尾部
mylist2 = [29, 33, 30]
mylist.extend(mylist2)
print(mylist)

# 取出第一个元素
print(f"第一个元素是{mylist[0]}")

# 取出最后一个元素
print(f"最后一个元素是{mylist[-1]}")

# 查找31元素下标
print(f"元素31的下标是{mylist.index(31)}")
```

数据容器：元组(tuple)

元组同列表一样，都是可以封装多个、不同类型的元素在内。但最大的不同点在于：**元组一旦定义完成，就不可修改**

定义语法：

```
# 定义元组字面量
(元素，元素，....)
# 定义元组变量
变量名称 = (元素，元素，....)
# 定义空元组
变量名称 = ()        # 法1
变量名称 = tuple()   # 法2
```

- 当定义单个元素的元组时，必须在定义后加上`,`，否则会自动将单个元素作为字符串类型进行处理。
- 元组只有index()，count()，len(元组)方法。
- 从元组中取出元素和列表中取出元素一样用`元组名[元素下标]`取出
- 元组的内容直接修改会报错，但是如果元组内含有list的内容，则可以修改

```
my_tuple = ('张三', 21, ["basketball", "rap"])

# 查询年龄所在下标位置
print(f"年龄所在下标位置为{my_tuple.index(21)}")

# 查询学生姓名
print(f"学生姓名为{my_tuple[0]}")

# 删除爱好一项
my_tuple[2].pop(0)
print(my_tuple)

# 添加一项爱好
my_tuple[2].append("jump")
print(my_tuple)
```

数据容器：字符串(str)

字符串是字符的容器，一个字符串可以存放任意数量的字符。

字符串同元组一样，是一个**无法修改的数据容器**

字符串常用方法：

- index("查找字符")：返回查找字符串的起始下标
- replace(字符串1，字符串2)：将字符串内的的全部字符串1，替换为字符串2，其不是修改字符串本身，而是得到了一个新字符串。
- slpit(分隔符字符串)：按照指定的分隔符字符串，将字符串划分为多个字符串，并存入列表对象中。**注：**字符串本身不变，而是得到了一个列表对象

```
my_str = "zhangsna, lisi, wangwu"
str_list = my_str.split(",")
print(str_list)
```

- strip()：`字符串.strip()`：去前后空格；`字符串.strip(字符串)`：去**前后**指定字符串，即将字符串前后的指定字符串去除

```
my_str = "   zhangsna, lisi, wangwu   "
my_str2 = my_str.strip()
print(my_str2)
my_str4 = "12zhangsna, lisi, wangwu21"
my_str3 = my_str4.strip("12")
print(my_str3)
```

```
zhangsna, lisi, wangwu
zhangsna, lisi, wangwu
```

序列

序列是指：内容连续、有序，可使用下标索引的一类数据容器。列表、元组、字符串，均可以视为序列

序列支持切片，即:列表、元组、字符串，均支持进行切片操作

切片:从一个序列中，取出一个子序列

语法：`序列[起始下标:结束下标:步长]`：表示从序列中，从指定位置开始，依次取出元素，到指定位置结束，得到一个新序列

- 起始下标表示从何处开始，可以留空，留空视作从头开始
- 结束下标(不含)表示何处结束，可以留空，留空视作截取到结尾
- 步长表示，依次取元素的间隔
  - 步长1表示，一个个取元素
  - 步长2表示，每次跳过1个元素取
  - 步长N表示，每次跳过N-1个元素取
  - 步长为负数表示，反向取(注意，起始下标和结束下标也要反向标记)

注意：此操作不会影响序列本身，而是会得到一个新的序列

数据容器：集合(set)

特点：

- 集合内不允许重复元素（去重）
- 集合内元素是无序（不支持下标索引）

基本语法：

```
# 定义字面量
{元素, 元素, ...}
# 定义集合变量
变量名称 = {元素, 元素, ...}
# 定义空集合
变量名称 = set() #只有一种方法，因为 变量名称 = {} 表示空字典的定义
```

集合的常用操作-修改

因为集合是无序的，所以集合不支持：下标索引访问，但是集合和列表一样，是允许修改的。

- 添加新元素

  语法：`集合.add(元素)`，将指定的元素添加到集合内

- 移除元素

语法：`集合.remove(元素)`，将指定的元素从集合内移除

- 随机取出一个元素

语法：`集合.pop()`，从集合中随机取出一个元素，同时集合本身被修改，元素被移除

- 清空集合

语法：`集合.clear()`，清空集合中所有元素

- 取两个集合的差集

语法：`集合1.difference(集合2)`，取出集合1和集合2的差集（集合1有而集合2没有的），得到一个新的集合，集合1和集合2不变

- 消除两个集合的差集

语法：`集合1.difference_update(集合2)`，对比集合1和集合2，在集合1内，删除和集合2相同的元素，集合1被修改，集合2不变

- 两个集合合并为一个集合

语法：`集合1.union(集合2)`，将集合1和集合2组合成新集合，得到新的集合，集合1和集合2不变

```
my_set = set()

my_list =["zhangsan", "lisi", "wamgwu", "zhangsan", "lisi", "mazi"]
for element in my_list:
    my_set.add(element)
print(my_set)
```

数据容器：字典(dict)

字典的定义，同样使用{}，不过存储的元素是一个个的**键值对**

语法：

```
# 字面量
{key: value, key: value, ...}
# 定义字典变量
变量名称 = {key: value, key: value, ...}
# 定义空字典
变量名称 = {}
变量名称 = dict{}
```

字典数据的获取

字典同集合一样，不可以使用下标索引，但是字典可以通过Key值来取得对应的value

语法：`字典[key]`可以取到对应的value

```
stu_dict = {"zhangsan": 100, "lisi": 29}
print(stu_dict[zhangsan])   # 结果100

# 定义嵌套dict
stu_score = {
    "zhangsan" : {
        "语文": 78,
        "数学": 99,
```

```
        "英语": 81
    }, "lisi": {
        "语文": 88,
        "数学": 79,
        "英语": 91
    },"wangwu": {
        "语文": 72,
        "数学": 94,
        "英语": 85
    }
}

print(stu_score)
# 遍历字典
for stu in stu_score:
    for obj in stu_score[stu]:
        print(stu_score[stu][obj])
    print(stu_score[stu])
print(stu_score["zhangsan"]["语文"])
```

字典的注意事项：

- 键值对的Key和Value可以是任意类型（Key不可为字典）
- 字典内的Key不允许重复，重复添加等同于覆盖原有数据
- 不可用下标检索，可以用key来检索value

字典的常用操作

- 新增元素

语法：`字典[key] = value`，结果：字典被修改，新增了元素

- 更新元素

语法：`字典[key] = value`，结果：字典被修改，元素被更新

- 删除元素

语法：`字典.pop(key)`，获取指定key的value，同时字典被修改，指定Key的数据被删除

- 清空字典

语法：`字典.clear()`，字典被修改，元素被清空

- 获取全部的key

语法：`字典.keys()`，返回一个dict_keys类型，包含所有的Key

- 获取字典长度

语法：`len(字典)`，计算字典内的元素数量

五类数据容器的总结对比

数据结构	元素数量	元素类型	下标索引	重复元素	可修改性	数据有序	使用场景
列表	支持多个	任意	支持	支持	是	可修改、可重	复的一批数据
元组	支持多个	任意	支持	支持	否	不可修改、可重	复的一批数据记
字符串	支持多个	仅字符	支持	支持	不支持	不可修改、可重	一串字符的记录
集合	支持多个	任意	不支持	不支持	支持	不可修改、可重	不可重复的数据
字典	Key:Value	Key：除字典外任意类型, Value：任意类型	不支持	不支持	支持	不可修改、可重	以Key检索Value

数据容器的通用操作

- `len(数据容器)`：返回容器中元素个数
- `max(数据容器)`：返回容器中的最大元素
- `min(数据容器)`：返回容器中的最小元素
- `list(容器), str(容器), tuple(容器), set(容器)`，具有下标索引可以通用类型转换
- `sorted(容器, [reverse = True])`：对容器中元素排序并返回，默认从小到大排序，排序的结果会通通转换为list

文件操作

文件的编码

UTF-8 是目前全球通用的编码格式

文件的读取操作

open()打开函数

在python中使用open函数，可以打开一个已经存在的文件，或者创建一个新文件，语法：

`文件对象 = open(name, mode, encoding)`

name：需要打开的目标文件名

mode：打开文件的模式：只读、写入、追加等

encoding：编码格式（推荐使用UTF-8）

mode常用的三种基础访问模式

模式	描述
`'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。
`'w'`	打开一个文件只用于写入。如果该文件已存在则内容会被覆盖（即文件被截断为零长度，然后从开头开始写入）。如果该文件不存在，创建新文件。
`'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。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。
`'x'`	创建并打开一个文件用于写入。如果该文件已存在，则操作失败。
`'b'`	二进制模式（可以与以上模式结合使用，如 `'rb'` 或 `'wb'`）。用于读写二进制文件。
`'t'`	文本模式（默认值，通常可省略）。
`'+'`	更新（读取和写入）模式。需要与其他模式结合使用，如 `'r+'`、`'w+'` 或 `'a+'`。

read()方法

语法：`文件对象.read(num)`，num表示要从文件中读取的数据的长度（单位是字节），如果没有传入num，那么就表示读取文件中所有数据，如果在一个程序中多次调用read()方法，则下一次的read()方法的起始位置是上一次read()的结束位置，返回str类型

readlines()方法

语法：`文件对象.readlines()`，按照行的方式把整个文件中的内容进行一次性读取，并且返回的是一个列表，其中每一行的数据为一个元素，readlines()同样会受到read()指针的影响

注：在readlines()函数调用后，每一行的末尾会有一个换行符`\n`，如果使用print函数打印的话会多一个换行符，此时需要使用`line = line.strip()  # 消除每一行的/n`来消除每一行的换行符

readline()方法

语法：`文件对象.readline()`，一次读取一行内容

for循环读取文件行

```
for line in open("D:/aaa.txt", "r", encoding="UTF-8"): # 取出的line类型为str类型
    print(line)
```

close()方法

语法：`文件对象.close()`，关闭文件对象，如果不调用close，同时程序没有停止运行，那么这个文件将一直被python程序占用

with open 语句

语法：

```
with open(name, mode, encoding) as f:
    f.readlines()
```

通过with open的语句块中对文件进行操作，可以在操作完成后自动关闭close文件，避免遗忘掉close方法

文件的写入操作

语法：

```
# 1、打开文件
f = open("python.txt","w")

# 2、文件写入
f.write("hello world")

# 3、内容刷新
f.flush()
```

注：

- 直接调用write，内容并未真正写入文件，而是会积攒在程序的内存中，称之为缓冲区
- 当调用flush的时候，内容会真正写入文件，close方法内置flush功能
- 这样做是避免频繁的操作硬盘，导致效率下降(攒一堆，一次性写磁盘)

文件的追加操作

语法：

```
# 1、打开文件，通过a模式打开即可
f = open("python.txt", "a")

# 2、文件写入
f.write("hello world")

# 3、内容刷新
f.flush()
```

注：

- a模式，文件不存在会创建文件
- a模式，文件存在会在最后，追加写入文件

python异常

捕获常规异常

语法：

```
try:
    可能发生错误的代码
except[异常 as 别名]:
    如果出现异常执行的代码
```

捕获指定异常

语法：

```
try:
    print(name)
except NameError as e:
    print("name变量名称未定义错误")
```

注：

- 如果尝试执行的代码的异常类型和要捕获的异常类型不一致，则无法捕获异常
- 一般try下方只放一行尝试执行的代码

捕获多个异常

```
try:
    print(name)
except (NameError, ZeroDivisionError) as e:
    print("name变量名称未定义错误")
```

捕获所有异常

```
try:
    可能存在异常的语句
except Exception as e:
    异常处理
```

异常的else和finally语法

else表示的是如果没有异常要执行的代码

```
try:
    可能存在异常的语句
except Exception as e:
    异常处理
else:
    没有异常时要处理的代码
```

finally表示的是无论是否异常都要执行的代码，例如关闭文件

```
try:
    可能存在异常的语句
except Exception as e:
    异常处理
else:
    没有异常时要处理的代码
finally:
    无论如何都要执行的代码
```

异常的传递

异常时具有传递性的，当函数2调用函数1，而函数1中发生异常，并且没有捕获处理这个异常的时候，异常会传递到函数2，这就是**异常的传递性**，当所有函数都没有捕获异常的时候，程序就会报错。

Python模块

什么是模块：

Python 模块(Module)，是一个 Python 文件，以.py 结尾，模块能定义函数，类和变量，模块里也能包含可执行的代码。

模块的作用：python中有很多各种不同的模块，每一个模块都可以帮助我们快速的实现一些功能，比如实现和时间相关的功能就可以使用time模块我们可以认为一个模块就是一个工具包，每一个工具包中都有各种不同的工具供我们使用进而实现各种不同的功能。

大白话：模块就是一个Python文件，里面有类、函数、变量等，我们可以拿过来用(导入模块去使用)。

模块的导入方式：

模块在使用前需要先导入，导入的语法：

[from 模块名] import [模块 | 类 | 变量 | 函数 | *] [as 别名]

常用组合形式：

- import 模块名
- from 模块名 import 类、变量、方法等
- from 模块名 import *
- import 模块名 as 别名
- from 模块名 import 功能名 as 别名

例：导入time模块

```
# 导入时间模块
import time

print("begin")
# 让程序阻塞一秒
time.sleep(1)
print("end")

# from 模块名 import 功能名
from time import sleep

print("begin")
# 让程序阻塞一秒
sleep(1)
print("end")

# 使用 * 导入time模块的全部功能
from time import *

print("begin")
# 让程序阻塞一秒
sleep(1)
print("end")
```

自定义模块

调用自定义模块时，正常创建.py文件并导入即可

当导入多个模块的时候，且模块内有同名功能，当调用这个同名功能的时候，调用到的是后面导入的模块的功能，即后面的模块会将前面的模块覆盖掉

如：

```python
# 模块一的代码
def test(a, b):
    print(a + b)

# 模块二的代码
def test(a, b):
    print(a - b)

# 导入模块
from module1 import test
from module2 import test

# test函数是模块二中的函数
test(1, 2)
```

__main__变量

当只是在模块中想要调用，而在导入模块后的代码中并不想执行某些语句的时候使用`if __name__ == '__main__':`来实现导入后不再强制执行，否则一旦导入模块，模块中的所有语句都会强制执行

```python
def my_test(a, b):
    print(a + b)

if __name__ == '__main__':
    my_test(2, 3)
```

__all__变量

如果一个模块文件中有"__all__"变量，当使用`from xxx import *`导入时，只能导入这个列表中的元素

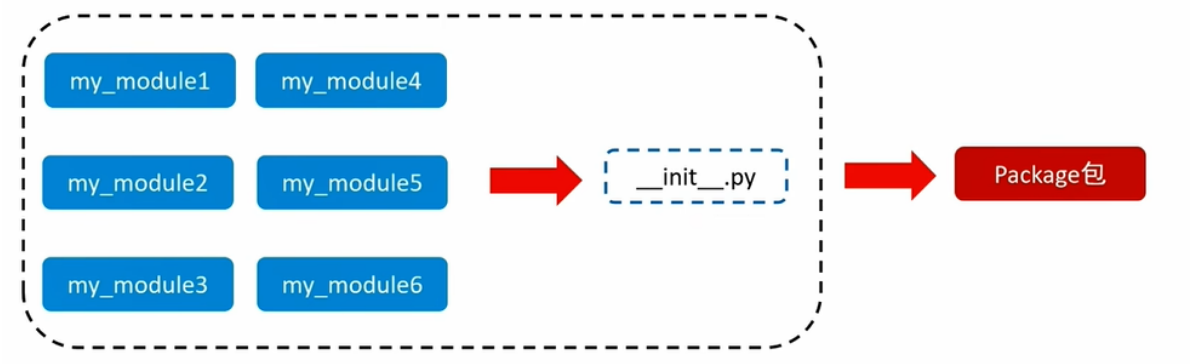
```python
__all__ = ["my_test"]

def my_test(a, b):
    print(a + b)

def my_test2(a, b):
    print(a - b)

# 主程序
from my_module import * # 只能使用此语法`*`时all变量才有效

my_test(1, 2)   # 不会报错
my_test2(1, 2)  # 会报错
```

python包

从物理上看，包就是一个文件夹，在该文件夹下包含了一个 __ init __.py 文件，该文件夹可用于包含多个模块文件。

从逻辑上看，包的本质依然是模块。

包的作用：
当模块文件越来越多时，包可以帮助我们管理这些模块，包的作用就是包含多个模块，但包的本质依然是模块



创建步骤：

1、新建包"my_package"

2、新建包内模块："my_module1"，"my_module2"

pycharm中创建步骤：

New -> python package -> 输入包名 -> 新建功能模块（pycharm会自动创建__ init __ .py文件）

导入语法：`import 包名.模块名`、`from 包名 import 模块名`、`from 包名.模块名 import 功能名`

__ all __变量

必须在"__ init _.py"文件中添加"__ all __ = []"，控制允许导入的模块列表

```
# __init__.py文件
__all__ = ["my_module1"]

# 调用
from my_package import *

my_module1.test1(1, 2)  # 正常调用
my_module2.test2(1, 2)  # 无法调用
```

第三方包

在python程序的生态中，有许多第三方包，可以帮助提高开发效率，如：

- 科学计算中常用的：numpy包
- 数据分析中常用的：pandas包
- 大数据计算中常用的：pyspark、apache-flink包
- 图形可视化常用的：matplotlib、pyecharts包
- 人工智能常用的：tensorflow包
- 等

但是由于是第三方包，python没有内置，所以需要安装导入后才能使用

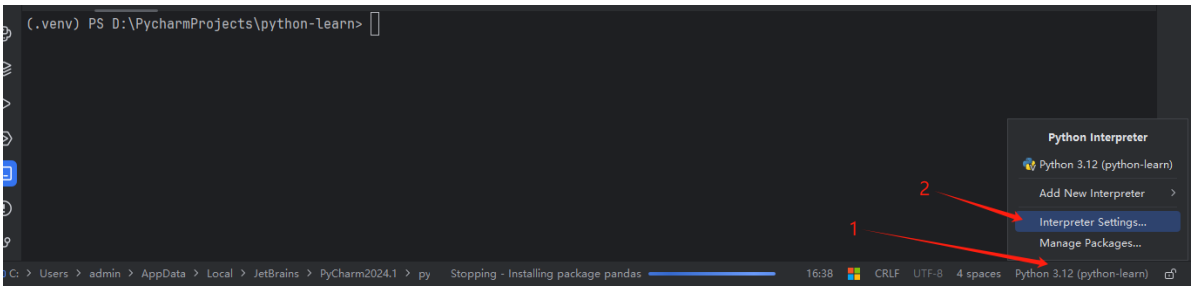
安装第三方包-pip

直接使用命令提示符，键入：`pip install 包名称`即可通过网络快速安装第三方包

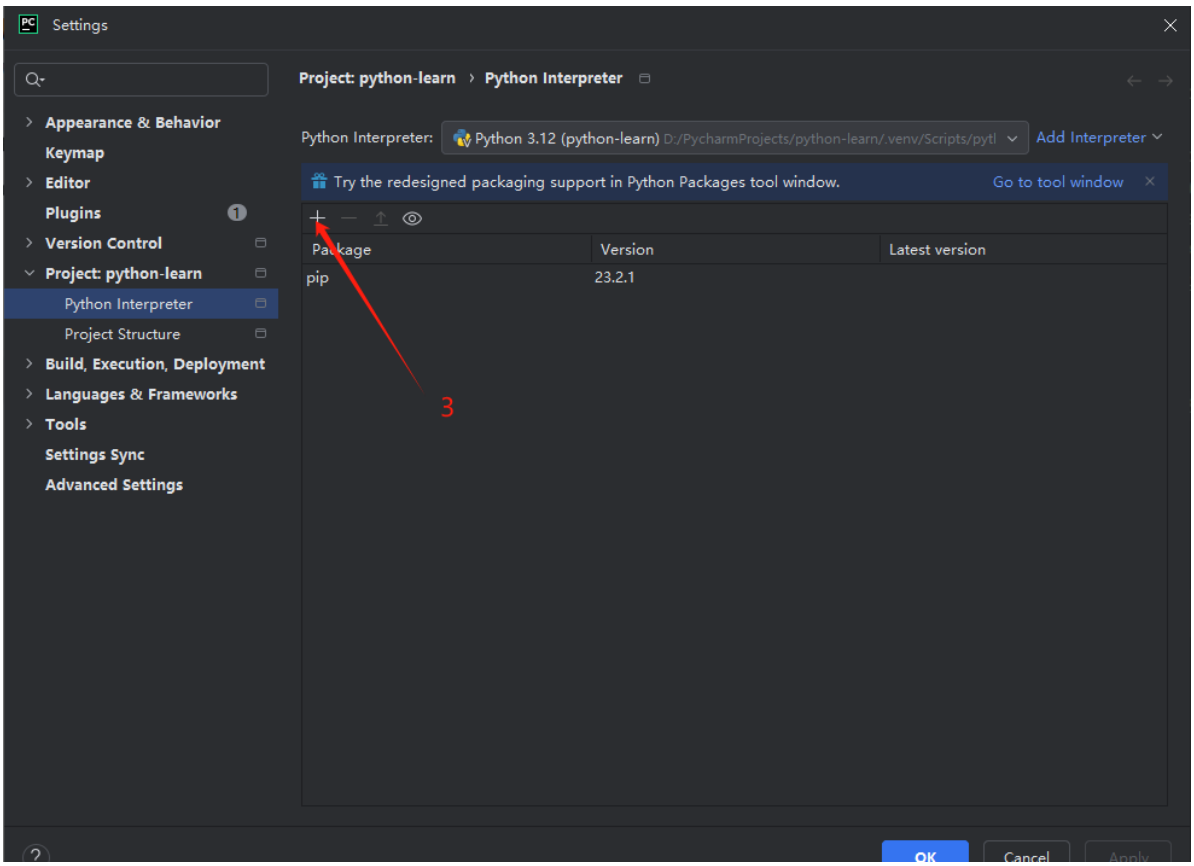
由于pip默认连接的是国外的镜像，可以通过`pip install -i https://pypi.tuna.tsinghua.edu.cn/simple 包名称`使用国内镜像进行下载

pycharm安装：

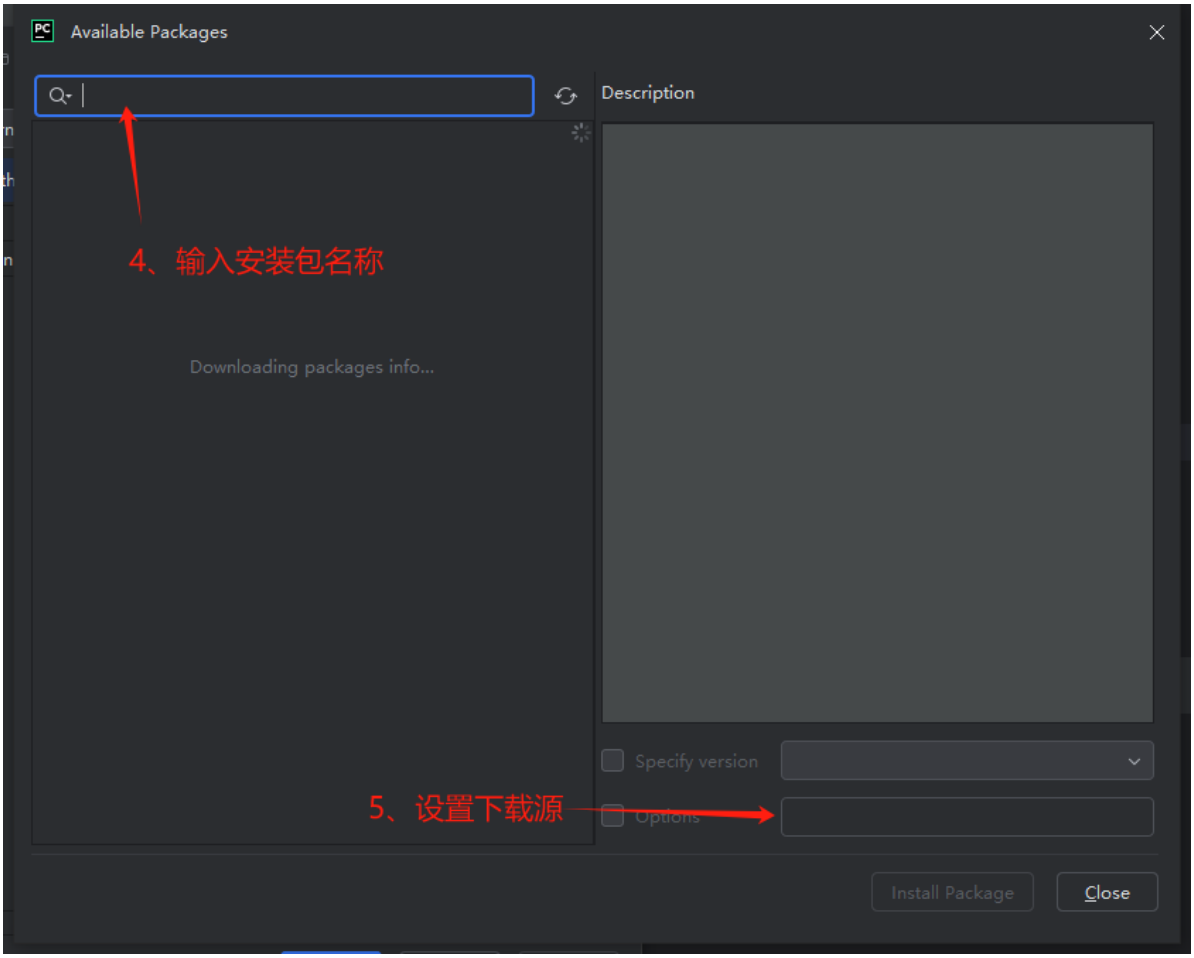
1、


(.venv) PS D:\PycharmProjects\python-learn>
Python Interpreter
Python 3.12 (python-learn)
Add New Interpreter
Interpreter Settings...
Manage Packages...
2
1
C: > Users > admin > AppData > Local > JetBrains > PyCharm2024.1 > py
Stopping - Installing package pandas
16:38
CRLF
UTF-8
4 spaces
Python 3.12 (python-learn)

2、


Settings
Project: python-learn › Python Interpreter
Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Project: python-learn
Python Interpreter
Project Structure
Build, Execution, Deployment
Languages & Frameworks
Tools
Settings Sync
Advanced Settings
Python Interpreter: Python 3.12 (python-learn) D:/PycharmProjects/python-learn/.venv/Scripts/pytl
Add Interpreter
Try the redesigned packaging support in Python Packages tool window.
Go to tool window
Package
Version
Latest version
pip
23.2.1
3
OK
Cancel
Apply

3、



综合实例

```
# 创建包名my_utils
# 创建模板str_util.py
def str_reverse(s):
    """
    字符串倒置
    :param s: 需要倒置的字符串
    :return: 倒置后的字符串
    """
    return s[::-1]


def substr(s, x, y):
    """
    截取字符串
    :param s: 需要截取的字符串
    :param x: 起始地址
    :param y: 结束地址，截取后不包含结束地址元素
    :return: 截取后的子串
    """
    return s[x: y: 1]

# 创建模板file_util.py
def print_file_info(file_name):
    """
    打开指定文件，并输出显示
    :param file_name: 指定文件名
    :return:
    """
```

```
    f = None
    try:
        f = open(file_name, "r", encoding="UTF-8")
        for line in f.readlines():
            print(line)
    except Exception as e:
        print(f"文件存在异常, 原因是{e}")
    finally:
        if f:   # 如果变量是None，表示false
            f.close()


def append_to_file(file_name, data):
    """
    指定文件并追加data
    :param file_name: 文件名
    :param data: 追加的data
    :return:
    """
    f = open(file_name, "a", encoding="UTF-8")
    f.write(data)
    f.close()


# 主程序
from my_utils.str_util import *
import my_utils.file_util
print(f"倒置'hello world'后：{str_reverse("hello world")}")
print(f"截取'hello world'后：{substr("hello world", 2, 6)}")

my_utils.file_util.print_file_info("D:/aaa.txt")
my_utils.file_util.append_to_file("D:/aaa.txt", "\n你好")
```

python数据和json数据的相互转化

```
# 导入json模块
import json

# 准备符合格式json格式要求的Python数据
data = [{"name":"zhangsan","age":16},{"name":"lisi","age":54}]

# 通过json.dumps(data)方法把 python数据转化为json数据,ensure_ascii=False表示转换为unicode字符格式
data = json.dumps(data, ensure_ascii=False)

# 通过json.loads(data)方法把json数据转化为python数据，最后转换为列表还是字典是看json的格式自动转换
data = json.loads(data)
```

pyecharts

导入pyecharts：`pip install pyecharts`

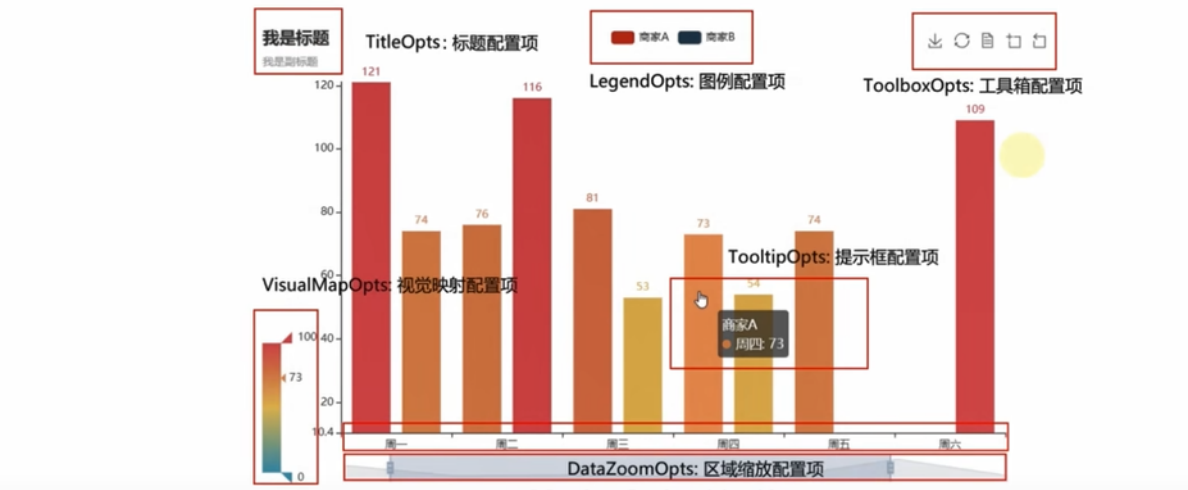
查看官方示例：官方示例

构建基础折线图

```
# 导入折线图line功能
from pyecharts.charts import Line

# 得到折线图对象
line = Line()

# 添加x轴
line.add_xaxis(["北京", "上海", "广州"])

# 添加y轴
line.add_yaxis("GDP", [15, 29, 31])

# 生成图表
line.render()
```

set_global_opts方法

- 这里**全局配置选项**可以通过set_global_opts方法来进行配置，相应的选项和选项的功能如下：




```
# 使用对象前需要导入对应的包
from pyecharts.options import TitleOpts

# 全局设置
line.set_global_opts(
    title_opts=TitleOpts(title="GDP展示")
)
```

具体全局变量可以在官方文档中查看

对象

类的创建：

```
class 类名:
    属性 = None
    属性 = None
    属性 = None
    ...
```

成员变量和成员方法

什么是类的行为（方法）：

```
class Student:
    name = None
    age = None
    def say(self):
        print({self.name})
```

可以看出，类中：

- 不仅可以定义属性用来记录数据
- 也可以定义函数，用来记录行为

其中：

- 类中定义的属性（变量），称之为：成员变量
- 类中定义的行为（函数），称之为：成员方法

成员方法的定义语法

```
def 方法名(self, 形参1, ..., 形参N):
    方法体
```

可以看到，在方法定义的参数列表中，有一个：self关键字

self关键字是成员方法定义的时候，必须填写的

- 它用来表示类对象自身的意思
- 当我们使用类对象调用方法的是，self会自动被python传入
- **在方法内部，想要访问类的成员变量，必须使用self**

构造方法

python类中可以使用：__init__()方法，称之为构造方法

可以实现：

- 在创建类对象（构造类）的时候，会自动执行
- 在创建类对象（构造类）的时候，将传入参数自动传递给__init__方法使用

```
class Student:
    name = None
    age = None # 可以在构造函数中声明，同时省略属性的定义

    def __init__(self, name, age):
        self.name = name
        self.age = age
        print(f"创建了一个student对象，对象姓名:{self.name}")


stu = Student("张三", 24)
```

魔术方法

前面的_ _init_ _方法，是python类内置的方法之一，这些内置的类方法，各自有各自特殊的功能，这些内置的方法称之为：魔术方法

_ _ str _ _字符串方法

当类被输出时，默认输出的是地址，用_ _str_ _方法可以将对象以字符串形式输出

```
    # str魔术方法
    def __str__(self):
        return f"对象名字{self.name}，对象年龄{self.age}"

print(stu.__str__())
print(stu)
print(str(stu)) # 三种语句效果输出一样
```

_ _lt_ _小于符号比较方法

```
    def __lt__(self, other):
        return self.age < other.age

stu = Student("张三", 24)
stu1 = Student("张三", 23)
print(stu.__lt__(stu1))
print(stu > stu1)
print(stu < stu1)
```

_ _le_ _小于等于比较符号方法

可用于：<=，>=两种比较运算符上

```
def __le__(self, other):
        return self.age <= other.age
```

_ _eq_ _比较运算符方法

```
def __eq__(self, other):
        return self.age == other.age
```

_ _ getitem_ _方法

此方法返回指定key相关的value（`__getitem__`就是get item翻译就是获取item）

对于list来说，key是index

对于dict来说就是设定的key和value

```
class Tag:
    def __init__(self, id):
        self.id = id

    def __getitem__(self, item):
        print("调用__getitem__")
        return self.id

a = Tag("This is id")
print(a.id)
print(a['python'])
```

输出：

```
This is id #打印了a.id

# 下面是调用了__getitem__方法
调用__getitem__
This is id
```

私有成员

定义私有成员的方式非常简单：

- 私有成员变量：变量名以__开头（2个下划线）
- 私有成员方法：方法名以__开头（2个下划线）

```
class Student:
    name = None
    age = None
    __sex = None    #私有变量

    def __printName():
        print("my name")    #私有方法
```

使用私有变量

私有成员无法被类对象使用，但是可以被其他的成员使用

```
class Phone:
    __run_voltage = 0.5

    def __print_voltage(self):
        print("以单核模式运行")

    def call_by_5g(self):
        if self.__run_voltage >= 1.0:
            print("以5g模式运行")
        else:
            self.__print_voltage()


phone = Phone()
phone.call_by_5g()
```

私有成员的实际意义：

在类中提供仅供内部使用的属性和方法，而不对外开放（类对象无法使用）

继承

语法：

```
class 类名(父类名):
    属性
    ...
    方法
    ...
```

多继承

python的类之间也支持多继承，即一个类，可以继承多个父类

语法：

```
class 类名(父类1, 父类2, ..., 父类n):
    类内容体
```

`pass`关键字：没有任何意思，当不需要在类中写内容体，为了满足语法要求，通过pass关键字来补全语法，功能是让语法不产生错误

注：多继承中，如果父类有同名方法或属性，先继承的优先级高于后继承

复写父类的成员属性和成员方法

直接进行重写即可

调用父类同名成员

一旦复写父类成员，当对象调用该成员时，会调用复写后的新成员，如果需要使用被复写的父类的成员，需要特殊的调用方式：

方式1：

- 调用父类成员
  - 使用成员变量：`父类名.成员变量`
  - 使用成员方法：`父类名.成员方法(self)`

方式2：

- 使用super()调用父类成员
  - 使用成员变量：`super().成员变量`
  - 使用成员方法：`super().成员方法()`

类型注解

当我们调用方法，进行传参的时候（快捷键ctrl + p弹出提示）：

当调用的是内置模块的时候，会提示输入的参数类型，而对于自定义方法传参时，仅能提示参数个数，而不能提示该参数的类型，这是就需要使用类型注解才能提示参数的对应数据类型

类型注解支持：

- 变量的类型注解
- 函数（方法）形参列表和返回值的类型注解

变量的类型注解

基础语法：`变量: 类型`

```
# 基础类型的类型注解
var_1: int = 19
var_2: float = 1.2
var_3: bool = True

# 类对象类型注解
class Student:
    pass

stu: Student = Student()

# 基础容器类型注解
my_list: list=[1, 2, 3]
my_tuple: tuple=(1, 2, 3)
my_set: set={1, 2, 3}
my_dict: dict={"youxin":666}
my_str: str ="youxin"

# 容器类型详细注解
my_1ist: list[int]=[1, 2, 3]
my_tuple: tuple[str, int, bool]=("youxin", 666,True)
my_set: set[int]={1, 2, 3}
my_dict: dict[str, int]={"youxin":666}
```

注：

- 元组类型设置类型详细注解，需要将每一个元素都标记出来
- 字典类型设置类型详细注解，需要2个类型，第一个是key第二个是value

一般无法直接看出变量类型之时，会添加变量的类型注解

变量的类型注解语法：

- 语法1：`变量: 类型`
- 语法2：在注释中，`# type: 类型`

函数和方法类型注解

形参注解

函数和方法的形参类型注解语法：

```
def 函数方法名(形参名: 类型, 形参名: 类型, ......):
    pass
```

返回值注解

函数和方法的返回值类型注解语法：

```
def 函数方法名(形参: 类型, ...) -> 返回值类型:
    pass
```

Union联合类型注解

当容器类型中包含多种数据类型的时候，原本的类型注解就并不能完全表示所有的类型注解，这时就需要使用Union进行联合类型注解

语法：`容器名: 容器类型[Union[数据类型, 数据类型, ...]]`

例如：

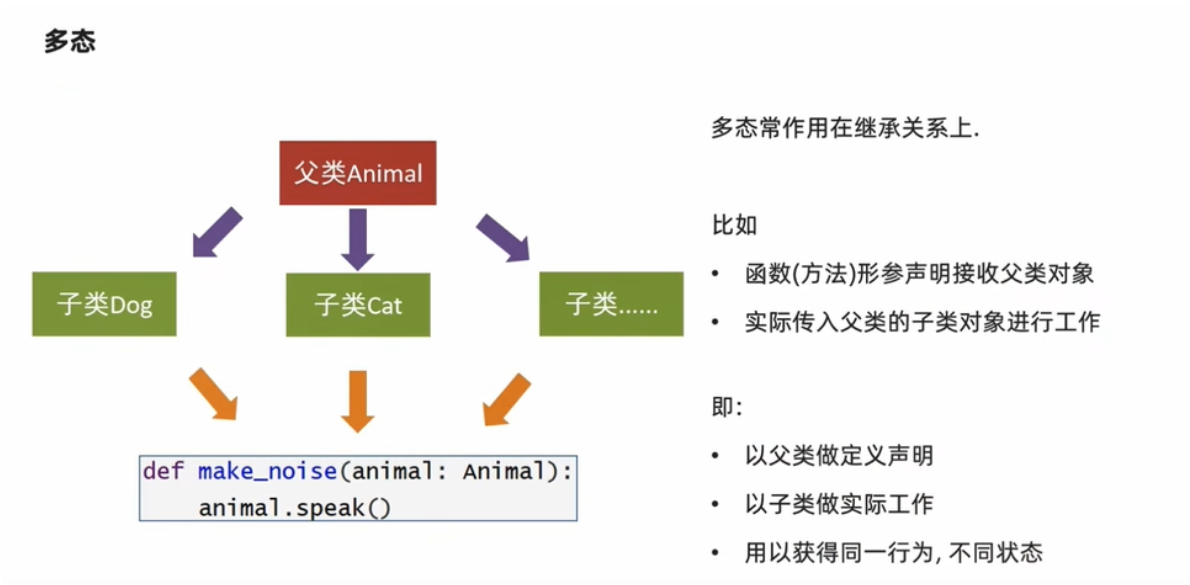
```
from typing import Union

my_list: list[Union[str, int]] = ["zhangsan", 22, "lisi"]

my_dict: dict[str, Union[int, str]] = {"name": 13}
```

注：Union联合类型注解，在变量注解、函数（方法）形参和返回值注解中，均可使用

```
def func(data: Union[str, int]) -> Union[str, int]:
    pass
```

多态

多态，指的是：多种状态，即完成某个行为时，使用不同的对象会得到不同的状态




```
class Animal:
    def speak(self):
        pass


class Dog(Animal):
    def speak(self):
        print("111")
```

```
class Cat(Animal):
    def speak(self):
        print("222")


dog = Dog()
cat = Cat()
dog.speak()
cat.speak()
```

抽象类（接口）

上面示例中：父类Animal的speak方法，是空实现，这种设计的含义是：

- 父类用来确定有哪些方法
- 具体的实现方法由子类自行决定

这种写法，就叫做抽象类，也可称之为接口

抽象类：含有抽象方法的类称之为抽象类

抽象方法：方法体是空实现的（pass）称之为抽象方法

SQL

Python使用MySQL

在python中，使用第三方库：pymysql来完成对MySQL数据库的操作

安装：`pip install pymysql`

创建到MySQL的数据库连接：

```
from pymysql import Connection
# 创建连接
conn = Connection(
    host='localhost',
    port=3306,
    user='root',
    passwd='1234',
)

# 打印mysql软件信息
print(conn.get_server_info())

# 关闭连接
conn.close()
```

执行非查询性质的sql语句：

```
from pymysql import Connection
# 创建连接
conn = Connection(
    host='localhost',
    database='pzhu',
    port=3306,
```

```
    user='root',
    passwd='1234',
)
# 获取游标对象
cursor = conn.cursor()

# 选择数据库
# conn.select_db("pzhu")

# 使用游标对象，执行sql语句
cursor.execute('create table test_table(id int, info varchar(255))')

# 打印mysql软件信息
print(conn.get_server_info())

# 关闭连接
conn.close()
```

执行查询性质的SQL语句

```
from pymysql import Connection
# 创建连接
conn = Connection(
    host='localhost',
    database='pzhu',
    port=3306,
    user='root',
    passwd='1234',
)
# 获取游标对象
cursor = conn.cursor()

# 选择数据库
# conn.select_db("pzhu")

# 使用游标对象，执行sql语句
cursor.execute('select * from employee')

# 获取查询结果
results: tuple = cursor.fetchall()
for result in results:
    print(result)

# 打印mysql软件信息
print(conn.get_server_info())

# 关闭连接
conn.close()
```

如何执行SQL查询

通过连接对象调用cursor()方法，得到游标对象

- 游标对象.execute()执行SQL语句
- 游标对象.fetchall()得到全部的查询结果封装入元组内

commit提交

在创建连接后，只是使用游标对象.excute()执行sql语句后并不能直接马上更改数据库中的内容

pymysql在执行数据插入或者其他产生数据更改的SQL语句时，默认是需要提交更改的，即需要通过确认这种更改行为

通过`连接对象.commit()`即可确认此行为

```
from pymysql import Connection
# 创建连接
conn = Connection(
    host='localhost',
    database='pzhu',
    port=3306,
    user='root',
    passwd='1234',
)
# 获取游标对象
cursor = conn.cursor()

# 执行sql
num = cursor.execute("insert into employee(lastName) values('zhangsan')")

# 通过commit确认
conn.commit()

print(f"影响的行数有{num}行")
# 关闭连接
conn.close()
```

自动commit

如果不想手动commit确认，可以在构建链接对象的时候，设置自动commit的属性

```
conn = Connection(
    host='localhost',
    database='pzhu',
    port=3306,
    user='root',
    passwd='1234',
    autocommit=True      # 设置自动提交
)
```