

Python 培训文档

Python 培训文档

1. 安装 Python

1.1 Windows

2. 安装和使用包管理工具 pip

2.1 安装 Python 包

2.2 升级 Python 包

2.3 卸载 Python 包

3. 基本 Python 使用

3.1 编写和运行 Python 脚本

3.2 交互式 Python Shell

4. Python 基础语法

4.1 print用法

4.1.1 基本用法

4.1.2 输出多个值

4.1.3 格式化输出

4.1.4 使用 f-string 格式化字符串 (Python 3.6+)

4.1.5 控制输出结束符

4.1.6 控制输出分隔符

4.1.7 输出到文件

4.1.8 清空缓冲区

4.1.9 输出不换行

4.1.10 格式化输出数字

4.1.11 输出到标准错误流

4.2 变量和数据类型

4.2.1 变量赋值:

4.2.2 基本赋值

4.2.3 多重赋值

4.2.4 同时为多个变量赋相同的值

4.2.5 交换变量的值

4.2.6 全局变量和局部变量

4.2.7 变量命名规则

4.2.8 变量类型

4.2.9 删除变量

4.3 控制流

4.3.1 与 (and)

4.3.2 或 (or)

4.3.3 非 (not)

逻辑操作符的优先级

总结

4.4 函数

4.4.1 函数的基本定义

4.4.2 示例

4.4.3 参数

4.4.3.1 位置参数

4.4.3.2 默认参数

4.4.3.3 可变参数

4.4.4 返回值

4.4.5 作用域

4.4.6 文档字符串

4.4.7 函数作为参数

4.4.8 匿名函数 (Lambda 函数)

4.4.9 函数装饰器

4.4.10 总结

4.5 数据结构

4.5.1 列表 (Lists)

4.5.2 元组 (Tuples)

4.5.3 字典 (Dictionaries)

4.5.4 集合 (Sets)

操作和方法

4.6 包

4.6.1 包的基本概念

4.6.2 包的创建和使用示例

4.6.3 高级用法

5. Python 的类和对象

类 (Class)

对象 (Object)

类和对象的详细关系

总结

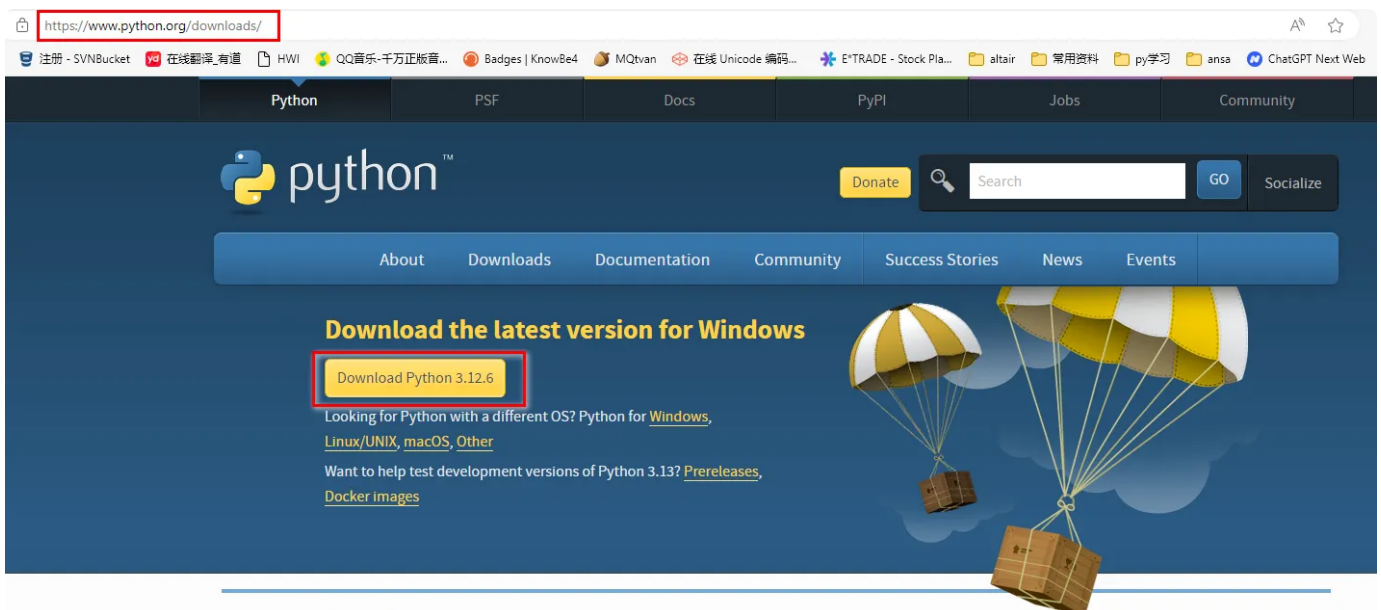
Python 培训文档

1. 安装 Python

1.1 Windows

1. 下载 Python:

- 访问 [Python 官方下载页面](#)，下载适用于 Windows 的最新版本 Python 安装包（通常是 `.exe` 文件）。




2. 运行安装程序:

- 双击下载的 `.exe` 文件，启动安装程序。

- 勾选 “Add Python to PATH” 选项。
- 点击 “Install Now” 或 “Customize Installation” 以进行自定义安装。

3. 验证安装:

- 打开命令提示符 (CMD) , 输入 `python --version` 或 `python3 --version` , 查看 Python 版本号。如果正确显示版本号, 安装成功。



```
命令提示符
Microsoft Windows [版本 10.0.19045.4780]
(c) Microsoft Corporation。保留所有权利。

C:\Users\yqxiang>python --version
Python 3.11.2

C:\Users\yqxiang>a
```

2. 安装和使用包管理工具 `pip`

Python 自带 `pip` 包管理工具, 用于安装和管理 Python 包。

2.1 安装 Python 包

- 使用 `pip` 安装包:

```
1 pip install package_name
```

例如, 安装 `requests` 包:

```
1 pip install requests
```

2.2 升级 Python 包

- 使用 `pip` 升级包:

```
1 pip install --upgrade package_name
```

2.3 卸载 Python 包

- 使用 `pip` 卸载包：

```
1 pip uninstall package_name
```

3. 基本 Python 使用

3.1 编写和运行 Python 脚本

1. 编写脚本：

- 使用文本编辑器（如 VSCode、PyCharm 或任何你喜欢的编辑器）编写 Python 脚本。保存为 `.py` 文件，例如 `script.py`。

2. 运行脚本：

- 在终端或命令提示符中，导航到脚本所在目录，然后输入：

```
1 python script.py
```

或者：

```
1 python3 script.py
```

Sublime Text (UNREGISTERED)

工具(T) 项目(P) 首选项(N) 帮助(H)

script.py x tkdemo.tcl x hm-ribbon.xml x HwiHandle.tcl

```
1 # -*- coding: utf-8 -*-
2 # @Author: yqxiang
3 # @Date: 2024-09-12 14:38:36
4 # @Last Modified by: yqxiang
5 # @Last Modified time: 2024-09-12
6 print("Hello Python")
```

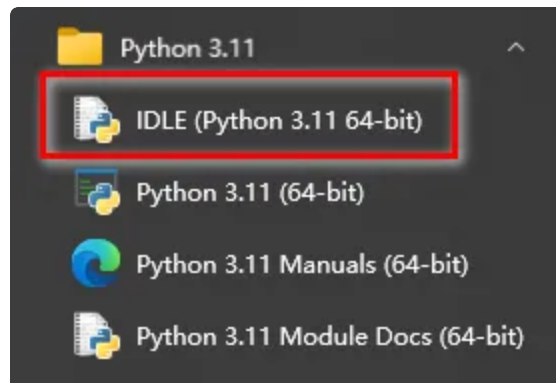
命令提示符

Microsoft Windows [版本 10.0.19045.4780]
(c) Microsoft Corporation。保留所有权利。

C:\Users\yqxiang>python C:\Users\yqxiang\Desktop\script.py
Hello Python

3.2 交互式 Python Shell

- 启动 Python Shell



- 你可以在提示符下直接输入 Python 代码并立即执行。

```
*IDLE Shell 3.11.2*
File Edit Shell Debug Options Window Help
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 AMD64] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print(
(*args, sep=' ', end='\n', file=None, flush=False)
Prints the values to a stream, or to sys.stdout by default.
```

4. Python 基础语法

4.1 print用法

`print()` 函数是 Python 中用于将输出打印到控制台的函数。下面是关于 `print()` 函数的一些详细用法说明：

```
print(
(*args, sep=' ', end='\n', file=None, flush=False)
Prints the values to a stream, or to sys.stdout by default.
```

4.1.1 基本用法

```
1 print("Hello, World!")
```

```
In [113]: print("Hello, World!")
Hello, World!
```

4.1.2 输出多个值

```
1 name = "Alice"
2 age = 30
3 print("Name:", name, "Age:", age)
```

```
In [112]: name = "Alice"
...: age = 30
...: print("Name:", name, "Age:", age)
Name: Alice Age: 30
```

4.1.3 格式化输出

```
1 name = "Bob"
2 age = 25
3 print("Name: {}, Age: {}".format(name, age))
```

```
In [114]: name = "Bob"
...: age = 25
...: print("Name: {}, Age: {}".format(name, age))
Name: Bob, Age: 25
```

4.1.4 使用 f-string 格式化字符串 (Python 3.6+)

```
▼ Python |
1 name = "Charlie"
2 age = 20
3 print(f"Name: {name}, Age: {age}")
```

```
In [115]: name = "Charlie"
...: age = 20
...: print(f"Name: {name}, Age: {age}")
Name: Charlie, Age: 20
```

4.1.5 控制输出结束符

```
▼ Python |
1 print("Hello", end=" ")
2 print("World")
```

```
In [116]: print("Hello", end=" ")
...: print("World")
Hello World
```

4.1.6 控制输出分隔符

```
1 print("a", "b", "c", sep=", ")
```

```
In [117]: print("a", "b", "c", sep=", ")
a, b, c
```

4.1.7 输出到文件

```
1 with open("output.txt", "w") as f:
2     print("Hello, World!", file=f)
```

4.1.8 清空缓冲区

```
1 import sys
2 print("Hello", end="")
3 sys.stdout.flush() # 清空缓冲区立即输出
```

4.1.9 输出不换行

```
1 print("Hello", end="")
2 print("World")
```

```
In [118]: print("Hello", end="")
...: print("World")
HelloWorld
```

4.1.10 格式化输出数字

```
1 num = 3.14159
2 print("Value of pi: {:.2f}".format(num))
```

```
In [119]: num = 3.14159
...: print("Value of pi: {:.2f}".format(num))
Value of pi: 3.14
```

4.1.11 输出到标准错误流

```
1 import sys
2 print("Error: Something went wrong", file=sys.stderr)
```

这些是一些常见的 `print()` 函数用法，有助于你更好地控制输出内容和格式。

4.2 变量和数据类型

4.2.1 变量赋值:

在 Python 中，变量赋值是`将一个值分配给变量的过程`。以下是关于 Python 变量赋值的详细说明：

4.2.2 基本赋值

```
1 x = 5
```

4.2.3 多重赋值

```
1 a, b, c = 5, 3.2, "Hello"
```

4.2.4 同时为多个变量赋相同的值

```
1 x = y = z = 0
```

4.2.5 交换变量的值

```
1 a, b = 5, 10
2 a, b = b, a
```

4.2.6 全局变量和局部变量

在函数内部，如果你想要修改全局变量的值，可以使用 `global` 关键字。

```
Python |
1 x = 10
2 y = 10
3 def modify_global():
4     global x
5     x = 20
6
7 def modify_y():
8     y = 20
9
10 modify_global()
11 modify_y()
12 print('x=',x) # 输出 20
13 print('y=',y) # 输出 10
```

```
In [123]: x = 10
...: y = 10
...: def modify_global():
...:     global x
...:     x = 20
...:
...: def modify_y():
...:     y = 20
...:
...: modify_global()
...: modify_y()
...: print('x=',x) # 输出 20
...: print('y=',y) # 输出 10
x= 20
y= 10
```

4.2.7 变量命名规则

变量名只能包含字母、数字和下划线。

变量名可以以字母或下划线开头，但不能以数字开头。

变量名对大小写敏感。

名称不要与python关键字冲突

以下是 Python 中变量名称不能使用的关键字（关键词）。这些名称在 Python 中有特殊含义，因此不能用作变量名或其他标识符：

关键字	说明
<code>False</code>	布尔值，表示假
<code>None</code>	表示空值或无值
<code>True</code>	布尔值，表示真
<code>and</code>	逻辑与运算符
<code>as</code>	用于别名导入或上下文管理
<code>assert</code>	断言，用于调试
<code>break</code>	跳出循环
<code>class</code>	定义类
<code>continue</code>	跳过当前循环的剩余部分，继续下一轮循环
<code>def</code>	定义函数
<code>del</code>	删除对象或变量
<code>elif</code>	条件语句中的“否则如果”
<code>else</code>	条件语句中的“否则”
<code>except</code>	捕获异常

<code>finally</code>	无论是否发生异常都执行的代码块
<code>for</code>	循环语句
<code>from</code>	从模块中导入特定部分
<code>global</code>	声明全局变量
<code>if</code>	条件语句
<code>import</code>	导入模块
<code>in</code>	成员运算符，检查值是否在序列中
<code>is</code>	身份运算符，检查对象是否相同
<code>lambda</code>	定义匿名函数
<code>nonlocal</code>	声明非局部变量
<code>not</code>	逻辑非运算符
<code>or</code>	逻辑或运算符
<code>pass</code>	空语句，表示什么都不做
<code>raise</code>	引发异常
<code>return</code>	从函数返回值
<code>try</code>	捕获异常的开始
<code>while</code>	循环语句
<code>with</code>	上下文管理器
<code>yield</code>	生成器函数中的返回值

当然，`list` 和 `dict` 等内置类型名称也有其特殊意义，虽然它们不是 Python 的关键字，但在某些情况下，使用这些名称作为变量名可能会引发问题。以下是

Python 的内置类型和一些常用对象的名称，使用这些名称作为变量名可能会导致不必要的混淆或覆盖内置功能：

内置名称	说明
<code>list</code>	列表类型，用于存储有序的集合
<code>dict</code>	字典类型，用于存储键值对
<code>set</code>	集合类型，用于存储唯一的元素
<code>tuple</code>	元组类型，用于存储不可变的有序集合
<code>str</code>	字符串类型，用于存储文本
<code>int</code>	整数类型
<code>float</code>	浮点数类型
<code>bool</code>	布尔类型
<code>range</code>	用于生成一个不可变的整数序列
<code>complex</code>	复数类型
<code>bytes</code>	字节类型
<code>bytearray</code>	可变字节类型
<code>memoryview</code>	内存视图类型
<code>object</code>	所有类的基类
<code>type</code>	类型对象
<code>NoneType</code>	<code>None</code> 的类型

注意事项

- **避免覆盖：**使用 `list`、`dict` 等作为变量名会覆盖原有的内置类型定义，从而影响代码的其他部分，导致潜在的错误或难以调试的问题。

- **命名规范**：建议使用更具描述性的名称来避免这种情况，例如 `my_list` 或 `data_dict`。

在编程过程中，尽量避免使用这些名称作为变量名，以保持代码的清晰性和可维护性。

```
In [132]: lst = list()

In [133]: list = 1

In [134]: lst = list()

-----
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4852\1736711593.py in <module>
----> 1 lst = list()

TypeError: 'int' object is not callable
INFO:ipykernel.inprocess.ipkernel:Exception in execute request:
-----
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4852\1736711593.py in <module>
----> 1 lst = list()

TypeError: 'int' object is not callable

In [135]: del list

In [136]: lst = list()
```

4.2.8 变量类型

在 Python 中，变量的类型是根据赋给它们的值自动确定的，这就是所谓的动态类型语言。例如：

```
1 int_x = 5 # 整数
2 float_y = 3.14 # 浮点数
3 str = "Hello" # 字符串
4 boolean = True # 布尔类型
```

4.2.9 删除变量

```
1 x = 5
2 del x # 删除变量 x
```

这些是关于 Python 变量赋值的一些基本和高级用法。

4.3 控制流

- 条件语句:

```
1 if x > 10:
2     print("x is greater than 10")
3 elif x == 10:
4     print("x is 10")
5 else:
6     print("x is less than 10")
```

- 循环:

```
1 # for loop
2 for i in range(5):
3     print(i)
4
5 # while loop
6 count = 0
7 while count < 5:
8     print(count)
9     count += 1
```

- 逻辑操作符:

在 Python 中，逻辑操作符用于执行布尔运算，它们分别是 与 (`and`)、或 (`or`) 和 非 (`not`)。以下是这些操作符的详细说明及示例：

4.3.1 与 (`and`)

`and` 操作符用于检查多个条件是否都为 `True`，只有当所有条件都为 `True` 时，结果才为 `True`。

```
Python |  
1 a = True  
2 b = False  
3  
4 result = a and b # False, 因为 b 是 False  
5 print(result)
```

```
In [1]: a = True  
...: b = False  
...:  
...: result = a and b # False, 因为 b 是 False  
...: print(result)  
False
```

- 示例：

```
1 x = 5  
2 y = 10  
3 z = 15  
4  
5 result = (x < y) and (y < z) # True, 因为 x < y 和 y < z 都为 True  
6 print(result) # 输出: True
```

```
In [2]: x = 5  
...: y = 10  
...: z = 15  
...:  
...: result = (x < y) and (y < z) # True, 因为 x < y 和 y < z 都为 True  
...: print(result) # 输出: True  
True
```

4.3.2 或 (`or`)

`or` 操作符用于检查多个条件中至少有一个为 `True`，只要有一个条件为 `True`，结果就为 `True`。

```
1 a = True
2 b = False
3
4 result = a or b # True, 因为 a 是 True
5 print(result)
```

```
In [4]: a = True
...: b = False
...:
...: result = a or b # True, 因为 a 是 True
...: print(result)
True
```

- 示例：

```
1 x = 5
2 y = 10
3 z = 3
4
5 result = (x < y) or (y < z) # True, 因为 x < y 为 True, 尽管 y < z 为 False
6 print(result) # 输出: True
```

```
In [5]: x = 5
...: y = 10
...: z = 3
...:
...: result = (x < y) or (y < z) # True, 因为 x < y 为 True, 尽管 y < z 为 False
...: print(result) # 输出: True
True
```

4.3.3 非 (`not`)

`not` 操作符用于反转布尔值。即，如果原值是 `True`，使用 `not` 后变为 `False`，反之亦然。

```
1 a = True
2
3 result = not a # False, 因为 a 是 True
4 print(result)
```

```
In [6]: a = True
...:
...: result = not a # False, 因为 a 是 True
...: print(result)
False
```

- 示例：

```
1 x = 5
2
3 result = not (x < 10) # False, 因为 x < 10 为 True, 使
  用 not 后变为 False
4 print(result) # 输出: False
```

逻辑操作符的优先级

逻辑操作符的优先级从高到低是：

1. not
2. and
3. or

- 示例：

```
1 x = 5
2 y = 10
3 z = 15
4
5 result = not (x < y) and (y < z) # 首先计算 not (x <
  y), 然后计算 and 操作
6 print(result) # 输出: False, 因为 not (x < y) 是 False
```

总结

- `and`：用于检查多个条件是否都为 `True`。
- `or`：用于检查多个条件中至少有一个为 `True`。
- `not`：用于反转布尔值。

这些逻辑操作符是构建复杂条件判断和控制流程的重要工具。

4.4 函数

- 定义和调用函数:

```
1 def greet(name):  
2     return f"Hello, {name}!"  
3  
4 print(greet("Alice"))
```

在 Python 中，`def` 关键字用于定义函数。函数是可重用的代码块，可以执行特定的任务。使用函数可以提高代码的可读性和可维护性。下面是对 `def` 的详细解析，包括函数的定义、参数、返回值、作用域等。

4.4.1 函数的基本定义

使用 `def` 关键字定义函数，后面跟着函数名和参数列表，然后是冒号 (`:`)，接着是函数体。

```
1 def function_name(parameters):  
2     # 函数体  
3     # 执行的代码  
4     pass # 占位符，表示函数体为空
```

4.4.2 示例

```
1 def greet(name):  
2     print(f"Hello, {name}!")  
3  
4 greet("Alice") # 输出: Hello, Alice!
```

4.4.3 参数

4.4.3.1 位置参数

位置参数是最常见的参数类型，调用函数时，参数的值根据位置传递。

```
1 def add(x, y):  
2     return x + y  
3  
4 result = add(5, 3) # result = 8
```

4.4.3.2 默认参数

可以为参数指定默认值，如果调用时未提供该参数，则使用默认值。

```
1 def greet(name="Guest"):  
2     print(f"Hello, {name}!")  
3  
4 greet() # 输出: Hello, Guest!  
5 greet("Alice") # 输出: Hello, Alice!
```

4.4.3.3 可变参数

使用 `*args` 和 `**kwargs` 可以接收可变数量的参数。

`*args` 用于接收位置参数的元组。

`**kwargs` 用于接收关键字参数的字典。

```

1 def variable_args(*args):
2     for arg in args:
3         print(arg)
4
5 variable_args(1, 2, 3) # 输出: 1 2 3
6
7 def variable_kwargs(**kwargs):
8     for key, value in kwargs.items():
9         print(f"{key}: {value}")
10
11 variable_kwargs(name="Alice", age=30) # 输出: name: A
    lice, age: 30

```

4.4.4 返回值

使用 `return` 语句返回函数的值。如果没有 `return`，函数将返回 `None`。

```

1 def multiply(x, y):
2     return x * y
3
4 result = multiply(4, 5) # result = 20

```

4.4.5 作用域

函数内部定义的变量是局部变量，只能在函数内部访问。全局变量在函数外部定义，可以在函数内部访问。

```

1  _x = 10  # 全局变量
2
3  def my_function():
4      y = 5  # 局部变量
5      print(_x)  # 可以访问全局变量
6      print(y)  # 可以访问局部变量
7
8  my_function()
9  # print(y)  # 会引发 NameError, 因为 y 是局部变量

```

4.4.6 文档字符串

可以使用文档字符串（docstring）为函数添加文档说明。文档字符串位于函数定义的第一行，用三个引号包围。

```

Python |
1  def add(x, y):
2      """返回 x 和 y 的和"""
3      return x + y
4
5  print(add.__doc__)  # 输出: 返回 x 和 y 的和

```

4.4.7 函数作为参数

函数可以作为参数传递给其他函数。

```

1  def apply_function(func, value):
2      return func(value)
3
4  def square(x):
5      return x * x
6
7  result = apply_function(square, 4)  # result = 16

```

4.4.8 匿名函数 (Lambda 函数)

Python 还支持使用 `lambda` 关键字定义匿名函数，这种函数通常用于简单的操作。

```
1 add = lambda x, y: x + y
2 result = add(3, 5) # result = 8
```

4.4.9 函数装饰器

装饰器是用于修改函数行为的高级功能。它们是函数的函数，可以在函数定义时应用。

```
1 def decorator_function(original_function):
2     def wrapper_function():
3         print("Wrapper executed before {}".format(origi
4             nal_function.__name__))
5         return original_function()
6     return wrapper_function
7 @decorator_function
8 def display():
9     print("Display function executed.")
10
11 display()
```

4.4.10 总结

- 使用 `def` 关键字定义函数。
- 函数可以接受多种类型的参数（位置参数、默认参数、可变参数）。
- 函数可以返回值，使用 `return` 语句。
- 理解作用域和局部变量与全局变量的区别。
- 使用文档字符串为函数提供说明。

- 函数可以作为参数传递，支持匿名函数和装饰器。

4.5 数据结构

在 Python 中，有几种内置的数据结构，包括列表（Lists）、元组（Tuples）、字典（Dictionaries）和集合（Sets）。这些数据结构在处理数据时非常有用，每种都有自己的特点和用途。下面是对这些数据结构的详细介绍：

4.5.1 列表（Lists）

- **定义：** 列表是一种有序、可变、允许重复元素的数据结构。用方括号 `[]` 定义，元素之间用逗号分隔。

```
1 my_list = [1, 2, 3, 'apple', 'banana']
```

- **特点：**
 - 可以通过索引访问元素，索引从 0 开始。
 - 支持切片操作。
 - 可以修改、添加或删除元素。

```
In [138]: my_list = [1, 2, 3, 'apple', 'banana']

In [139]: my_list[0]
Out[139]: 1

In [140]: my_list[1:3]
Out[140]: [2, 3]

In [141]: my_list.append(5)

In [142]: my_list
Out[142]: [1, 2, 3, 'apple', 'banana', 5]
```

4.5.2 元组（Tuples）

- **定义：** 元组是一种有序、不可变、允许重复元素的数据结构。用圆括号 `()`

定义，元素之间用逗号分隔。

Python |

```
1 my_tuple = (1, 2, 3, 'apple', 'banana')
```

- 特点：

- 不可变，一旦创建就不能修改。
- 可以通过索引访问元素，支持切片操作。

```
In [144]: my_tuple = (1, 2, 3, 'apple', 'banana')

In [145]: my_tuple[0]
Out[145]: 1

In [146]: my_tuple[1:3]
Out[146]: (2, 3)
```

4.5.3 字典 (Dictionaries)

- 定义：字典是一种无序、可变、键值对存储的数据结构。用大括号 `{}` 定义，每个键值对之间用冒号 `:` 分隔，键值对之间用逗号分隔。

```
1 my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

- 特点：

- 键值对存储数据，通过键访问值。
- 键必须是唯一的，值可以重复。
- 可以修改、添加或删除键值对。

```

In [148]: my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

In [149]: my_dict.keys()
Out[149]: dict_keys(['name', 'age', 'city'])

In [150]: my_dict.values()
Out[150]: dict_values(['Alice', 30, 'New York'])

In [151]: my_dict['name']
Out[151]: 'Alice'

In [152]: my_dict['sex'] = 'man'

In [153]: my_dict
Out[153]: {'name': 'Alice', 'age': 30, 'city': 'New York', 'sex': 'man'}

In [154]: del my_dict['sex']

In [155]: my_dict
Out[155]: {'name': 'Alice', 'age': 30, 'city': 'New York'}

In [156]: my_dict['name'] = 'Tom'

In [157]: my_dict
Out[157]: {'name': 'Tom', 'age': 30, 'city': 'New York'}

```

4.5.4 集合 (Sets)

- **定义：** 集合是一种无序、可变、不允许重复元素的数据结构。用大括号 `{}` 定义，元素之间用逗号分隔。

```
1 my_set = {1, 2, 3, 'apple', 'banana', 3, 4}
```

- **特点：**
 - 不允许重复元素，自动去重。
 - 支持集合运算，如并集、交集、差集等。

```
In [172]: my_set = {1, 2, 3, 'apple', 'banana', 3, 4}

In [173]: my_set
Out[173]: {1, 2, 3, 4, 'apple', 'banana'}

In [174]: new_set = {1, "red"}

In [175]: my_set - new_set
Out[175]: {2, 3, 4, 'apple', 'banana'}

In [176]: my_set | new_set
Out[176]: {1, 2, 3, 4, 'apple', 'banana', 'red'}

In [177]: my_set & new_set
Out[177]: {1}

In [178]: my_set ^ new_set
Out[178]: {2, 3, 4, 'apple', 'banana', 'red'}
```

操作和方法

- 共同点:

- 支持迭代操作。
- 可以使用 `len()` 函数获取长度。
- 可以使用 `in` 和 `not in` 检查元素是否存在。

- 列表方法:

- `append()`: 在列表末尾添加元素。
- `insert()`: 在指定位置插入元素。
- `remove()`: 移除指定元素。
- `pop()`: 弹出指定位置的元素。
- 等等。

```
In [180]: lst = [0,1,2]

In [181]: lst.append(4)

In [182]: lst
Out[182]: [0, 1, 2, 4]

In [183]: lst.insert(0,"new_add")

In [184]: lst
Out[184]: ['new_add', 0, 1, 2, 4]

In [185]: lst.remove(1)

In [186]: lst
Out[186]: ['new_add', 0, 2, 4]
```

- 字典方法：

- `get()`：获取指定键的值。
- `keys()`：返回所有键。
- `values()`：返回所有值。
- `items()`：返回所有键值对。
- 等等。

- 集合方法：

- `add()`：添加元素。
- `remove()`：移除指定元素。
- `union()`：返回两个集合的并集。
- `intersection()`：返回两个集合的交集。
- 等等。

```
In [197]: my_set
Out[197]: {2, 3, 4}

In [198]: my_set.add(1)

In [199]: my_set
Out[199]: {1, 2, 3, 4}

In [200]: my_set.remove(4)

In [201]: my_set
Out[201]: {1, 2, 3}

In [202]: new_set = {3,4}

In [203]: my_set.union(new_set)
Out[203]: {1, 2, 3, 4}

In [204]: my_set.intersection(new_set)
Out[204]: {3}
```

4.6 包

在 Python 中，包（Package）是组织模块（Module）的方式，用于管理大型程序的代码。包是一个包含多个模块的目录，这些模块通常具有相关的功能。以下是关于包的详细解释和用法：

4.6.1 包的基本概念

4.6.1.1 包的结构：

目录结构示例：

```
1  my_package/
2  |— __init__.py
3  |— module1.py
4  |— module2.py
5  |— sub_package/
6     |— __init__.py
7     |— sub_module1.py
```

- 包实际上是一个目录，其中包含一个特殊的文件 `__init__.py` 和其他模块文件（`.py` 文件）或子包目录。
- `__init__.py` 文件可以是空的，但它的存在表明该目录是一个包。它可以用来初始化包或执行包的初始化代码。

4.6.1.2 导入包中的模块：

- 从包中导入模块可以使用以下语法：

```
1 import my_package.module1
```

访问模块中的内容：

```
1 my_package.module1.some_function()
```

- 直接从模块中导入特定函数或类：

```
1 from my_package.module1 import some_function
2 some_function()
```

- 导入子包中的模块：

```
1 from my_package.sub_package import sub_module1
2 sub_module1.some_function()
```

4.6.1.3 `__init__.py` 文件：

- 这个文件的主要作用是初始化包，可以包含包的初始化代码。
- 可以在 `__init__.py` 中定义要导出的模块、函数或类，这样可以简化模块的导入。例如，在 `my_package/__init__.py` 中：

```
1 from .module1 import some_function
2 from .module2 import another_function
```

然后可以这样导入：

```
1 from my_package import some_function, another_function
```

4.6.2 包的创建和使用示例

1. 创建包：创建一个目录并在其中添加 `__init__.py` 文件及其他模块：

```
1 my_package/  
2 |— __init__.py  
3 |— greetings.py  
4 |— math_operations.py
```

`greetings.py` :

```
1 def say_hello():  
2     return "Hello, World!"
```

`math_operations.py` :

```
1 def add(a, b):  
2     return a + b
```

`__init__.py` :

```
1 from .greetings import say_hello  
2 from .math_operations import add
```

2. 使用包：在包外部的脚本或交互式环境中，可以这样导入和使用包中的模块：

```
1 from my_package import say_hello, add  
2  
3 print(say_hello()) # 输出: Hello, World!  
4 print(add(5, 3))   # 输出: 8
```

4.6.3 高级用法

1. 相对导入：在包内部的模块中，可以使用相对导入来引用其他模块。例如，

在 `sub_module1.py` 中引用 `module1` :

```
1 from ..module1 import some_function
```

2. **包的组织**: 包可以嵌套在其他包中, 形成层级结构。这使得组织和管理大型项目的代码变得更加灵活。
3. **动态导入**: 使用 `importlib` 模块可以在运行时动态导入模块:

```
1 import importlib
2 module = importlib.import_module('my_package.module1')
```

包是组织和管理 Python 代码的重要工具, 使得大型项目能够保持结构化和可维护。通过合理使用包和模块, 可以有效地组织代码并提升项目的可读性和可维护性。

5. Python 的类和对象

在 Python 中, 类 (Class) 和对象 (Object) 是面向对象编程 (OOP) 的核心概念。理解它们是掌握 Python 编程的重要步骤。以下是对类和对象的详细通俗介绍:

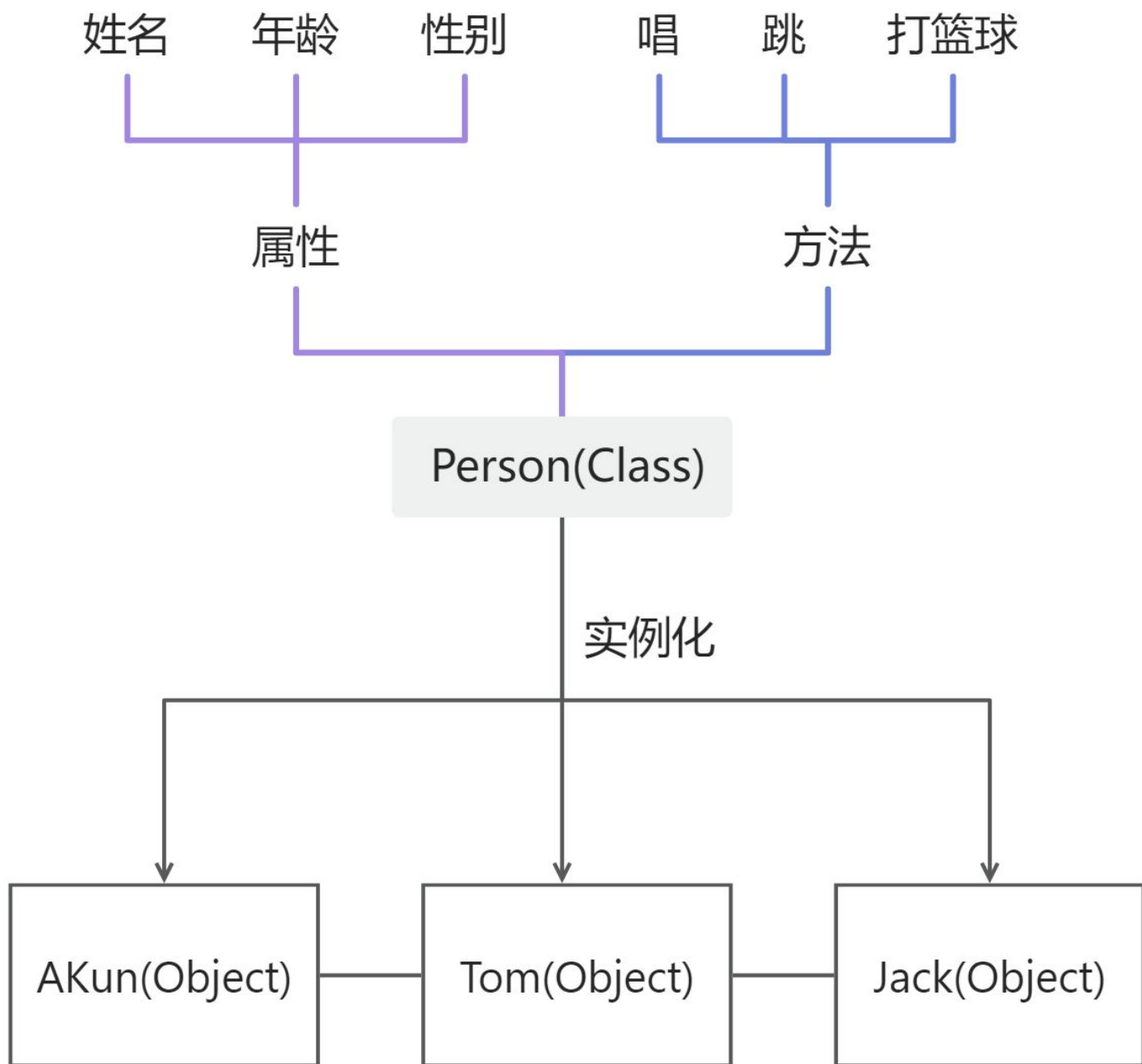
类 (Class)

概念:

- **定义**: 类是一个模板或蓝图, 用于创建对象。它定义了一类对象共有的属性和方法。属性是对象的特征, 方法是对对象的行为。

示例:

假设我们要创建一个描述人的类。我们可以定义一个 `Person` 类, 其中包含人的属性 (如名字、年龄、性别) 和方法 (如唱跳、打篮球)。



```
1 class Person:
2     def __init__(self, name, age, gender):
3         self.name = name      # 姓名
4         self.age = age        # 年龄
5         self.gender = gender  # 性别
6
7     def sing(self):
8         print(f"{self.name} is singing.") # 唱歌的方法
9
10    def dance(self):
11        print(f"{self.name} is dancing.") # 跳舞的方法
12
13    def rap(self):
14        print(f"{self.name} is rapping.") # rap的方法
15
16    def play_basketball(self):
17        print(f"{self.name} is playing basketball.")
18        # 打篮球的方法
```

- `__init__` 方法是构造函数，用于初始化对象的属性。`self` 代表当前实例。
- `sing` 和 `dance` 是方法，用于定义 `Person` 类对象的行为。

对象 (Object)

概念：

- **定义：**对象是类的实例。每个对象都有自己的属性值和可以调用的方法。对象是根据类创建的，类定义了对象的结构和行为。

示例：

使用 `Person` 类创建对象：

```
1 # 创建两个对象
2 p_1 = Person("AKun", 25, "Male")
3 p_2 = Person("Tom", 30, "Male")
4 p_3 = Person("Jack", 30, "Male")
```

- `p_1` 是 `Person` 类的一个实例。
- 对象 `p_1` 具有 `name`、`age` 和 `gender` 属性，且可以调用 `sing` 和 `dance` 方法。

操作对象：

```
1 p_1.sing()
2 p_2.dance()
```

类和对象的详细关系

1. 类是模板，对象是实例：

- 类定义了对应的结构（属性）和行为（方法）。
- 对象是类的实例，具有类定义的结构和行为，但每个对象的属性值可以不同。

2. 类的实例化：

```
1 p_1 = Person("AKun", 25, "Male")
```

- 创建对象的过程称为实例化。当你调用 `Person` 类时，Python 会创建一个新的 `Person` 对象实例。

3. 访问属性和方法：

- 通过点操作符 `.` 访问对象的属性和方法：

```
1 print(p_1.name) # 输出：AKun
2 print(p_1.sing()) # 输出：AKun is singing.
```

4. 封装：

- 封装是面向对象编程的一个特性，它将数据（属性）和方法（行为）捆绑在一起，并隐藏对象的内部细节。这使得对象的使用者不需要了解内部实现细节，只需知道如何使用对象的公共接口（方法）。

5. 继承和多态：

- **继承**：允许一个类继承另一个类的属性和方法，创建一个新的类。
- **多态**：允许不同的类用相同的接口调用不同的方法。

总结

- **类 (Class)**：定义了对象的结构和行为，类似于蓝图。
- **对象 (Object)**：类的实例，具有实际的数据和方法，可以操作这些数据。

通过理解类和对象，可以有效地使用 Python 的面向对象编程特性，将代码组织得更加清晰和可维护。