第一个Python程序

在系统学习Python前，我们先写一个基础的程序

```
print("Hello World")
```

运行后，应该能看到控制台输出

```
Hello World
```

在这个程序里，`print` 意为打印，用于向控制台输出，如下所示

```
print("Hello Beijing")
```

这段代码将会输出 `Hello Beijing`

变量和基本数据类型

变量的定义和使用

在Python中，变量用于存储数据。变量需要有它的数据类型。
定义变量的语法如下：

```
变量名 = 变量初始值
```

其中，变量名可以由字母、下划线和数字组成，下划线一般用于分隔单词，数字不能放在变量名的开头，如下表所示：

变量名	是否合法
a	是
1	否
_	是
hello_world	是
_i	是
hello world	否
Hello	是

以下代码展示了变量的定义：

```python
# 整数和浮点数
a = 10
b = 3.14

# 字符串
c = "Hello"

# 列表
d = [1, 2, 3, "four", 5.0]

# 元组
e = (1, 2, 3)

# 字典
f = {"name": "Alice", "age": 30}

print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
```

注：代码里的 `#` 符号表示它后面的内容是注释，是用于开发者理解程序的，机器在执行时不会管它。

这段代码定义了不同类别的变量，它的输出如下：

```
10
3.14
Hello
[1, 2, 3, 'four', 5.0]
(1, 2, 3)
{'name': 'Alice', 'age': 30}
```

基本数据类型

Python中有6大基本数据类型，分别为数字、字符串、列表、元组、集合、字典。这些数据类型有可变和不可变之分：

- 不可变（3 个）：数字、字符串、元组；
- 可变（3 个）：列表、字典、集合。

数字类型

数字类型包括整形（整数），浮点型（小数）等类型。数字类型可以进行加减乘除等运算：

```
a = 10
b = 3.14
print(a + b)  # 加法
print(a - b)  # 减法
print(a * b)  # 乘法
print(a / b)  # 除法
print(a ** 2)  # 乘方
print(a % 2)  # 取余
```

输出结果：

```
13.14
6.859999999999999
31.400000000000002
3.184713375796178
100
0
```

程序执行了相应的运算，但是我们发现，有一些结果并不精确，如 `6.859999999999999` ，在小数点后有许多位，这是因为计算机存储浮点数时的特性导致的，感兴趣的同学可以课后查阅资料，保证你们看不懂。

字符串

字符串由多个字符组成，单行字符串可以使用单引号、双引号表示，多行字符串可以使用三个单引号或三个双引号表示：

```
s1 = 'Hello, World!'
s2 = "I'm a Python programmer."
s3 = '''This is a multi-line string.
It can span multiple lines.
'''

print(s1)
print(s2)
print(s3)
```

输出：

```
Hello, World!
I'm a Python programmer.
This is a multi-line string.
It can span multiple lines.
```

字符串也可以进行多种操作，如下：

```python
s1 = 'Hello, World!'

# 加法操作：合并字符串
print(s1 + "!")

# 乘法操作：复制字符串多次
print(s1 * 2)

# 获得第3个字符
print(s1[2])

# 获取最后一个字符
print(s1[-1]])

# 获得第3到5个字符
print(s1[2: 5])
```

输出：

```
Hello, World!!
Hello, World!Hello, World!
l
!
llo
```

注意到，当我们想要获取字符串的第n个字符时，我们需要写 `s[n-1]` ，这意味着Python中，机器认为字符串的第1个字符应该是第0个字符。而当我们想要获取字符串的第m到n个字符时，我们需要写 `s[m-1: n]` （这种操作叫切片操作）这意味着n的那一位是不会被取到的。
在 `s[n]` 里，n称为索引，索引是从0开始计数的。
在 `s[m: n]` 里， `[m: n]` 称为切片，其包含m索引下的数据，但不包含n索引下的数据。

列表

列表由多个数据组成，因此列表中也能使用索引和切片。列表可以任意访问并修改这些数据：

```python
# 列表
lst = [1, 2, 3, 'four', 5.0]

# 访问列表中的元素
print(lst[0])
print(lst[0: 2])
print(lst[-1])

# 修改列表中的元素
lst[0] = 10
print(lst)
```

输出：

```
1
[1, 2]
5.0
[10, 2, 3, 'four', 5.0]
```

列表也可以进行增删查改操作，如下所示：

```python
# 创建一个列表
my_list = [1, 2, 3, 4, 5]

# 在列表末尾添加元素
my_list.append(6)
print("增加元素后:", my_list)

# 在指定位置插入元素
my_list.insert(2, 'two')  # 在索引为2的位置插入'two'
print("插入元素后:", my_list)

# 查找元素的索引
index_of_three = my_list.index(3)
print("元素3的索引是:", index_of_three)  # 注意，'two'插入后，3的索引变为2

# 修改指定索引位置的元素
my_list[2] = 'Two'  # 将索引为2的元素从'two'改为'Two'
print("修改元素后:", my_list)

# 删除指定索引位置的元素
del my_list[6]  # 删除索引为6的元素（即6）
print("删除元素后:", my_list)

# 使用remove()方法删除指定值的元素
my_list.remove(3)  # 删除值为3的元素
print("使用remove()删除元素后:", my_list)

# 使用pop()方法删除并返回指定索引位置的元素
popped_element = my_list.pop(1)  # 删除索引为1的元素（即2）并获取删除的元素
print("使用pop()删除的元素是:", popped_element)
print("pop()删除元素后:", my_list)

# 清除整个列表
my_list.clear()
print("清除列表后:", my_list)
```

输出:

```
增加元素后: [1, 2, 3, 4, 5, 6]
插入元素后: [1, 2, 'two', 3, 4, 5, 6]
元素3的索引是: 3
修改元素后: [1, 2, 'Two', 3, 4, 5, 6]
删除元素后: [1, 2, 'Two', 3, 4, 5]
使用remove()删除元素后: [1, 2, 'Two', 4, 5]
使用pop()删除的元素是: 2
pop()删除元素后: [1, 'Two', 4, 5]
清除列表后: []
```

元组

和列表相似，元组也可以存储多个数据，但是元组不能改变，即不能增删改：

```python
# 元组
tup = (1, 2, 3, 'four')

# 访问元组中的元素
print(tup[0])
```

输出：

```
1
```

集合

集合也可以存储多个数据，但和列表、元组不同，集合是无序的，且数据不能重复：

```python
# 集合
s = {1, 2, 3, 4, 4}  # 注意集合中的元素会自动去重
print(s)

# 添加元素到集合中
s.add(5)
print(s)
```

输出：

```
{1, 2, 3, 4}
{1, 2, 3, 4, 5}
```

注意：集合不能存储可变类型，如以下代码会报错：

```python
s = {1, 2, 3, 4, 4, []}
print(s)
```

报错如下：

```
Traceback (most recent call last):
  File "script.py", line 2, in
       s = {1, 2, 3, 4, 4, []}
TypeError: unhashable type: 'list'
```

字典

字典是无序可变的，它也可以存储多个数据，但是它可以根据键找到值：

```python
# 字典
d = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# 访问字典中的值
print(d['name'])

# 修改字典中的值
d['age'] = 31
print(d)

# 添加新的键值对
d['country'] = 'USA'
print(d)
```

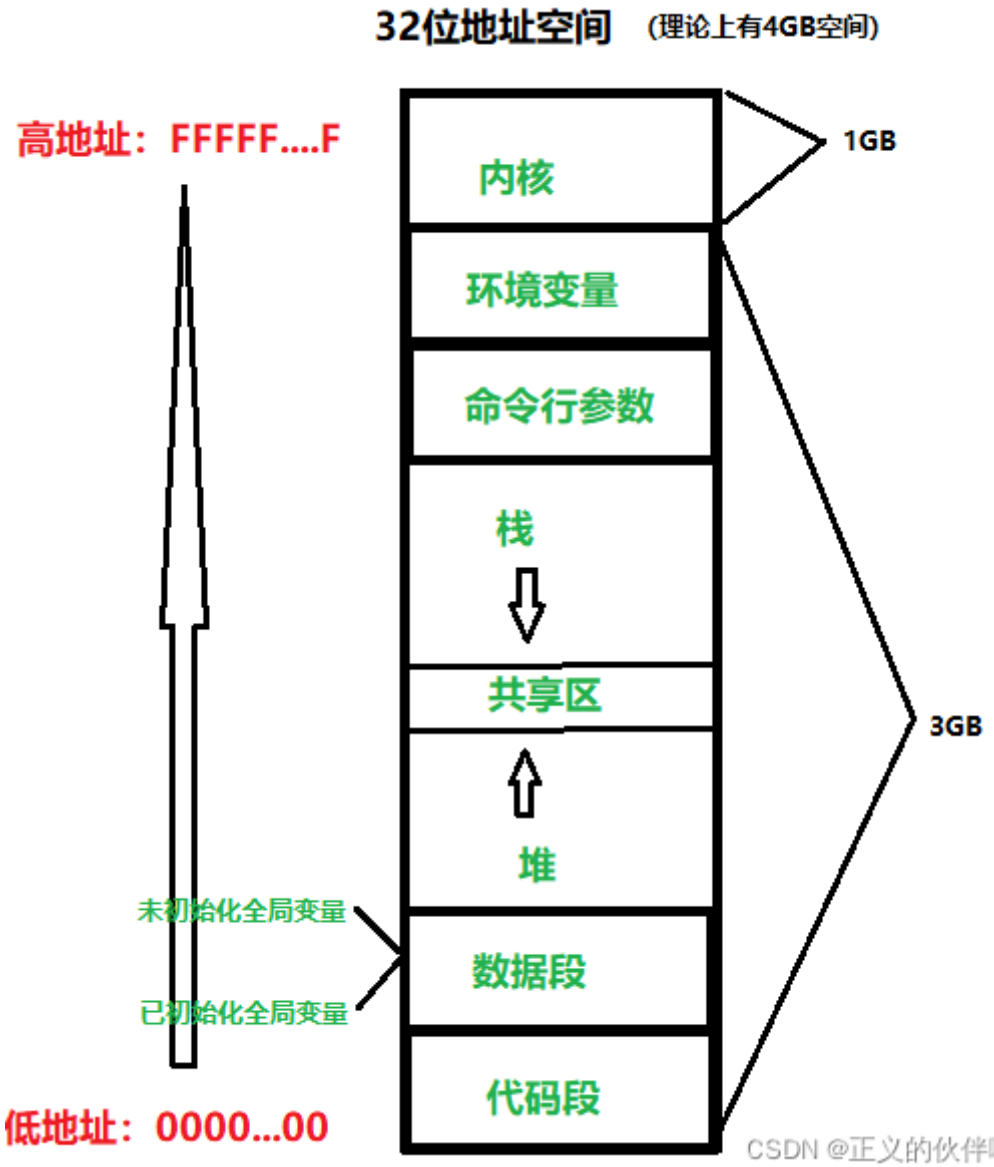
输出：

```
Alice
{'name': 'Alice', 'age': 31, 'city': 'New York'}
{'name': 'Alice', 'age': 31, 'city': 'New York', 'country': 'USA'}
```

变量在地址空间中的存储

计算机的任何进程运行时，操作系统都会给它一片虚拟的地址空间，进程可以在这片空间中存储数据。而当进程使用了一块空间后，计算机才会将其真正存储在内存中。

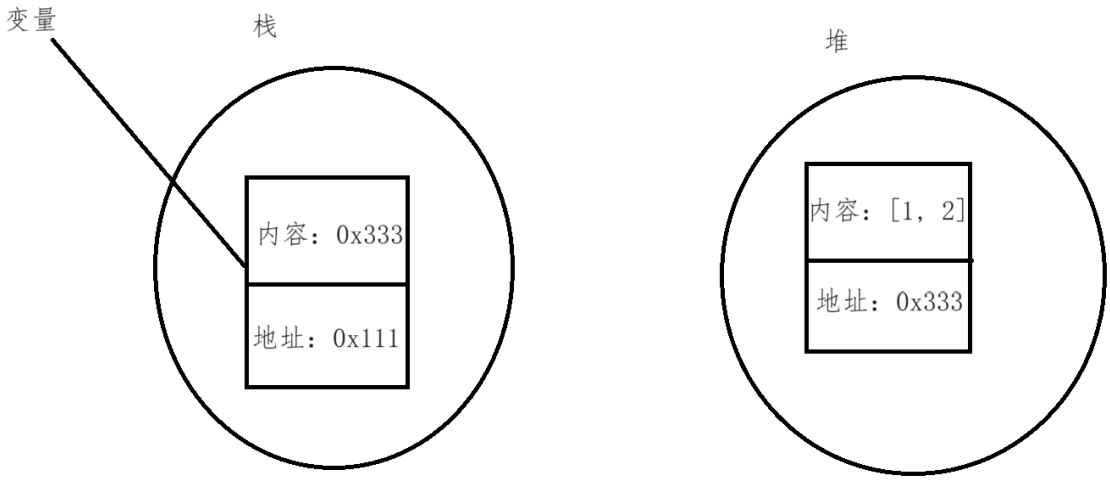
操作系统给计算机提供的地址空间分为很多部分，其主要部分为栈和堆，如图所示：



变量所存储的数据其实就是放在了地址空间中，但是对于可变和不可变的类型，变量有不同的存储方法。

可变类型

对于可变类型，Python会在堆中要来一块空间存储数据，然后在栈中要来一块空间存储堆中那块空间的地址，而用户定义的变量其实是在栈中，它所存储的是堆中的那块地址，如图所示：



以下代码检验了这一说法：

```
# Python在堆中存储了[1, 2, 3]，l1存储了这个列表的地址
l1 = [1, 2, 3]
# 将l2定义为l1存储的内容，即那个列表的地址
l2 = l1
# 通过操作l2操作那个列表
l2[1] = 3
# 打印l1，因为l2直接操作了列表的地址，因此输出[1, 3, 3]
print(l1)
```

输出：

```
[1, 3, 3]
```

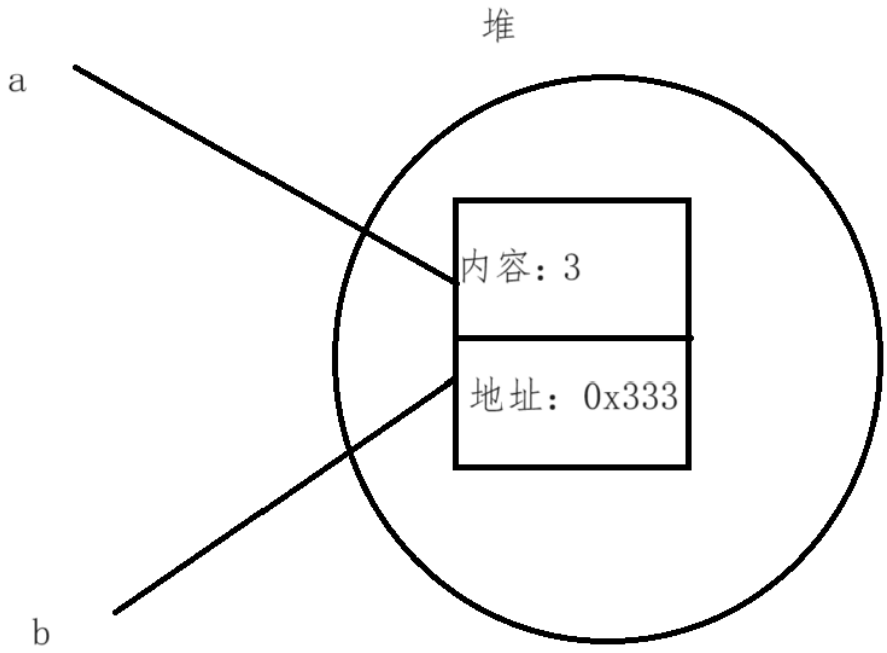
但是，由于操作变量相当于直接操作它存储的地址的数据，因此，通常称这个变量为对应数据的引用。

不可变类型

对于不可变类型，Python会创建一个公共的对象，创建变量时会创建这个公共的对象的引用。在执行完以下代码后，Python将会把a和b都设为一个空间的引用，而这个空间存储数据 `3` ：

```
a = 3
b = 3
```

如图所示：



类型转换

Python中，六大基本类型之间可以相互转换，如以下代码所示：

```python
# 数字转字符串
number = 123
number_to_str = str(number)
print(f"数字转字符串: {number} -> {number_to_str}")

# 字符串转数字
string_number = "123"
string_to_number = int(string_number)
print(f"字符串转数字: {string_number} -> {string_to_number}")

# 列表转元组
list_data = [1, 2, 3, 4, 5]
list_to_tuple = tuple(list_data)
print(f"列表转元组: {list_data} -> {list_to_tuple}")

# 元组转列表
tuple_data = (1, 2, 3, 4, 5)
tuple_to_list = list(tuple_data)
print(f"元组转列表: {tuple_data} -> {tuple_to_list}")

# 列表转集合
list_data_2 = [1, 2, 2, 3, 4, 4, 5, 5]
list_to_set = set(list_data_2)
print(f"列表转集合: {list_data_2} -> {list_to_set}")

# 集合转列表
set_data = {1, 2, 3, 4, 5}
set_to_list = list(set_data)
print(f"集合转列表: {set_data} -> {set_to_list}")
```

注：这里的 `f"XXX"` 用于字符串内插入数据。

这段代码将输出：

```
数字转字符串: 123 -> 123
字符串转数字: 123 -> 123
列表转元组: [1, 2, 3, 4, 5] -> (1, 2, 3, 4, 5)
元组转列表: (1, 2, 3, 4, 5) -> [1, 2, 3, 4, 5]
列表转集合: [1, 2, 2, 3, 4, 4, 5, 5] -> {1, 2, 3, 4, 5}
集合转列表: {1, 2, 3, 4, 5} -> [1, 2, 3, 4, 5]
```

type函数

type函数用于获得一个变量（对象）的类型，如下所示：

```
h = [1, 2, 3]
print(type(h))
```

这段程序将输出 `<class 'list'>` ，表示h是一个列表。

流程控制

条件语句

条件语句判断是否满足条件，并决定是否执行某段代码。其定义如下：

```
if 条件1:
    如果条件1为True，执行这里的代码
elif 条件2:
    如果条件1为False，且条件2为True，执行这里的代码
else:
    如果条件1和条件2都为False，执行这里的代码
```

其中， `elif` 可以不加，也可以加多个； `else` 可以不加。
举个例子：

```
age = 20

if age < 18:
    print("未成年人")
elif age >= 18 and age < 65:
    print("成年人")
else:
    print("老年人")
```

最终会输出：

```
成年人
```

循环语句

Python中，循环分为for循环和while循环。

for循环

for循环可以遍历一个可迭代对象（如列表、元组、字符串等），其形式如下：

```
for 变量 in 可迭代对象:
    循环体，对每个元素执行相应的操作
```

举个例子：

```
fruits = ['苹果', '香蕉', '橙子']

for fruit in fruits:
    print(fruit)
```

输出：

```
苹果
香蕉
橙子
```

while循环

while循环将会在满足条件时循环，不满足条件时退出，其形式如下：

```
while 条件:
    循环体，只要条件为True，就会一直执行
```

举个例子，我们可以用while计算阶乘（n! = n * (n - 1) * (n - 2) * ... * 1）：

```
n = 5
result = 1
while n > 0:
    result *= n
    n -= 1

print(result)
```

注：`*=`、`-=`表示将变量乘上（减去）一个数，并赋值到原来的变量上。
这段代码将会输出120，即5的阶乘。

循环控制语句

break

break用于跳出循环：

```
fruits = ['苹果', '香蕉', '橙子']

for fruit in fruits:
    if fruit == '香蕉':
        break
    print(fruit)
```

这将只输出 `苹果` ，因为遍历到香蕉时循环跳出了。

continue

continue用于跳过当前的迭代，进行下一次迭代。如果修改上面的代码：

```
fruits = ['苹果', '香蕉', '橙子']

for fruit in fruits:
    if fruit == '香蕉':
        continue
    print(fruit)
```

将会输出 `苹果` 和 `橙子` ，这是因为遍历到香蕉的时候，还没运行到 `print` 语句当前迭代就结束了。

比较运算符

在进行流程控制的时候，需要通过比较运算符判断条件是否成立，如下表所示：

符号	含义
==	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。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量 True 和 False 等价。
>=	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。

逻辑运算符

流程控制时同样需要逻辑运算符，如下表所示：

运算符	含义
and	与 - 判断两个表达式是否都满足
or	或 - 判断两个表达式是否有一个满足
not	非 - 如果x为真，则not x为假，反之为真

逻辑运算符的使用如以下代码所示：

```
a = 20
if not a < 0:  #如果a不小于0
    if a < 18:
        print("未成年")
    if a >= 18 and a < 65: # a大于等于18且a小于65，or的用法类似
        print("成年人")
    if a >= 65:
        print("老年人")
else:
    print("外星人")
```

函数

Python中，函数是一种数据类型。函数类型的变量是可调用对象。

函数的定义形式如下：

```
def 函数名(参数1, 参数2, ...):
    调用函数后运行的代码
```

其中，参数可以没有，也可以有多个，用逗号分开。

先通过以下代码了解一下函数：

```
def hello():
    print("Hello World")


h = hello
print(type(h))
print(callable(h))  # callable(h)，判断h是不是可调用对象
```

输出：

```
<class 'function'>
True
```

这说明函数是一个数据类型，同时是一个可调用对象。

函数的调用

函数调用时，只需要给函数变量加上小括号即可。在调用函数后，会执行函数体内的代码，如下：

```
def hello():
    print("Hello World")


h = hello
h()
```

然而，更常用的方法时直接使用函数名调用，如下：

```
def hello():
    print("Hello World")


hello()
```

这两段代码都将输出 `Hello World` ，这是因为 `hello` 函数中的 `print` 语句被执行了。

函数的参数

函数的参数相当于在调用函数时给函数传入的变量，函数可以对其进行操作，如下：

```
def hello(who):
    print(f"Hello {who}")


hello("Beijing")
```

这段程序将输出 `Hello Beijing` ，因为调用函数时，传入了 `Beijing` .

回调函数

如果将函数的参数也设为一个函数的话，传入函数的函数称为回调函数，如下：

```
def printer(func):
    func()


def hello():
    print(f"Hello World")


printer(hello)
```

在这段代码中，调用 `printer` 时，传入了 `hello` 作为参数，因此 `hello` 就是回调函数。这段代码将输出 `Hello World` 。

函数的返回值

函数执行完成后，可以用 `return` 语句，向调用者返回一个值，称为返回值，如下所示：

```
def add(x, y):
    return x + y


print(add(1, 2))
```

程序将输出 `3` 。这是因为 `add` 函数返回了1+2的值，作为 `print` 函数的参数。而 `print` 函数将其输出。

闭包函数

闭包函数指的是返回一个函数的函数，但是，这个被返回的函数必须使用闭包函数的变量，同时定义在闭包函数体内，如下：

```
def add_producer(x, y):
    def add():
        return x+y
    return add


add_func = add_producer(1, 2)
print(add_func())
```

程序将输出 `3` . 这是因为 `add_func` 是一个函数，而这个函数是 `add_producer` 返回的 `add` 函数，此时 `add` 函数就像这样：

```
def add():
    return 1+2
```

因此程序会输出 `3` .

类和对象

考虑我们要做一个宠物管理系统，需要存储宠物的各种数据，对宠物进行各种操作。具体要求如下：
存储数据：

- 宠物名称

- 宠物种类
- 宠物年龄

操作：

- 过了一年（宠物年龄+1）
- 向宠物打招呼（打印：你好，我<宠物年龄>岁的<宠物种类>，<宠物名称>）

结合字典和函数的相关知识，我们可以写出如下代码：

```
def init_pet(name, kind, age):
    return {"name": name, "kind": kind, "age": age}


# 过了一年
def spend_a_year(pet):
    pet["age"] += 1  # 因为pet是引用，所以可以直接操作


# 向宠物打招呼
def greet_to_pet(pet):
    print(f"你好，我{pet['age']}岁的{pet['kind']}，{pet['name']}")
```

这样的话，我们就可以在下面加上任意的操作代码了，比如可以加上：

```
my_pet = init_pet("金毛", "狗", 3)
greet_to_pet(my_pet)
spend_a_year(my_pet)
greet_to_pet(my_pet)
```

这将输出：

```
你好，我3岁的狗，金毛
你好，我4岁的狗，金毛
```

或者可以加上：

```
my_pet = init_pet("小金", "金鱼", 1)
spend_a_year(my_pet)
spend_a_year(my_pet)
greet_to_pet(my_pet)
```

这将输出：

```
你好，我3岁的金鱼，小金
```

总之，通过前面的一堆函数，我们可以进行任何相关的操作，增强了可扩展性。
但是，我们将函数（操作）和字典（数据）分开了，这就是所谓的面向过程思想。但是这样产生了一个问题，就是在调用函数前必须调用初始化函数，获得一个字典；调用函数时需要传入一个字典，这可能会很麻烦。
随着社会的发展，有人提出将操作和数据合在一起，形成一种数据类型。在这种类型的变量定义时，数据完成初始化。同时，可以直接调用这种类型的内部方法，进行操作。这种思想就是所谓的面向对象思想，这种数据类型就是类，这种数据类型的变量就是对象。
一个类的定义形式如下：

```
class 类名(继承的类):
    def __init__(self, 一大堆参数):
        初始化代码
    其他类内的函数定义
```

如果将前面的示例改写成类的形式，就像这样：

```python
class Pet:
    def __init__(self, name, kind, age):
        self.name = name
        self.kind = kind
        self.age = age

    # 过了一年
    def spend_a_year(self):
        self.age += 1

    # 向宠物打招呼
    def greet(self):
        print(f"你好，我{self.age}岁的{self.kind}，{self.name}")
```

类里面的self是调用该方法的对象本身。在3-5行中的 `self.XXX = XXX` 是在初始化字段，就像前面的代码的 `init_pet` 函数构建字典一样。
当类的变量被定义时， `__init__` 函数将被调用，除了self的其他参数都需要进行传递。
接下来我们将上面的使用代码改写一下，如下所示：

```python
my_pet = Pet("金毛", "狗", 3)
my_pet.greet()
my_pet.spend_a_year()
my_pet.greet()
```

```python
my_pet = Pet("小金", "金鱼", 1)
my_pet.spend_a_year()
my_pet.spend_a_year()
my_pet.greet()
```

类的继承

类的继承是指一个类大体上继承另一个类的字段（Python中并没有明确的字段定义）和方法（类内部的函数）。继承其他类的类叫子类，被继承的类叫父类。
子类可以在继承父类的基础上进行修改，如增加新方法、重写原来方法等。如以下代码所示：

```python
class Pet:
    def __init__(self, name, kind, age):
        self.name = name
        self.kind = kind
        self.age = age

    # 过了一年
    def spend_a_year(self):
        self.age += 1

    # 向宠物打招呼
    def greet(self):
        print(f"你好，我{self.age}岁的{self.kind}，{self.name}")


class Dog(Pet):
    # 父类中也有__init__方法，子类重新定义了一遍，子类的对象将使用子类的方法，称为重写
    def __init__(self, name, age):
        super().__init__(name, "狗", age)  # super：调用父类的对应方法


class GoldFish(Pet):
    # 父类中也有__init__方法，子类重新定义了一遍，子类的对象将使用子类的方法，称为重写
    def __init__(self, name, age):
        super().__init__(name, "金鱼", age)  # super：调用父类的对应方法
```

我们可以分别创建 `Dog` 类和 `GoldFish` 类的对象：

```python
dog = Dog("金毛", 3)
fish = GoldFish("小金", 1)
dog.greet()  # 子类继承了父类的greet方法
fish.greet()
```

这将输出：

```
你好，我3岁的狗，金毛
你好，我1岁的金鱼，小金
```

模块

模块的使用

Python提供了很多模块。可以使用 `import` 语句导入模块，然后用符号 `.` 使用模块中的函数、类或变量：

```python
# 导入math模块
import math

# 使用math模块中的sqrt函数
print(math.sqrt(16))
```

也可以通过 `as` 指定别名：

```python
# 导入math模块并为其指定别名
import math as m

# 使用别名访问sqrt函数
print(m.sqrt(25))
```

如果只想使用 `sqrt` 函数，可以使用 `from math import sqrt` ，如下：

```python
# 导入math模块中的sqrt函数
from math import sqrt

# 直接调用sqrt函数
print(sqrt(36))
```

模块的自定义

可以通过新建Python文件自定义模块，如下所示：

```
# 文件名：printer.py
print("this is printer")  # 这行代码将在import时执行


def printer(text):
    print(text)
```

```
# 文件名：main.py
import printer

printer.printer("Hello World")
```

运行main.py，程序会输出：

```
this is printer
Hello World
```

写在最后

这篇教程只是介绍了Python中非常基础的东西，在具体做项目时，可以使用包括但不限于百度、CSDN、Gitee、ChatGPT、文心一言等互联网工具查询项目所需的技术知识。