# Day 2：Python 四大数据类型总结



## 1. 基本数据类型

### 1.1 数值型

Python 中的数据皆是对象，比如被熟知的 int 整型对象、float 双精度浮点型、bool 逻辑对象，它们都是单个元素。举两个例子。

前缀加 `0x`，创建一个十六进制的整数：

```
0xa5 # 等于十进制的 165
```

使用 `e` 创建科学计数法表示的浮点数：

```
1.05e3 # 1050.0
```

### 1.2 容器型

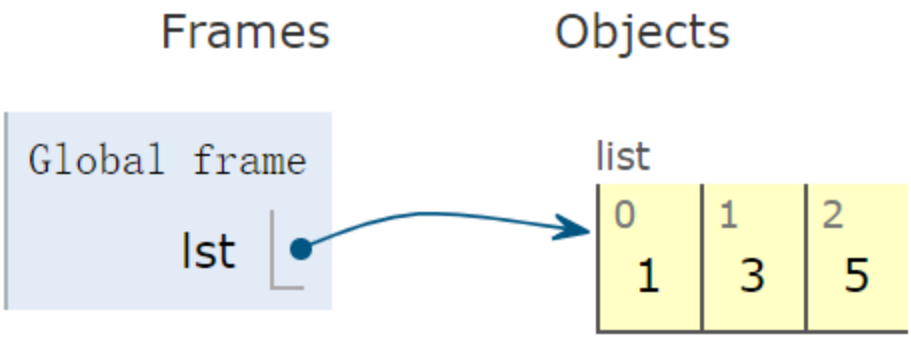
可容纳多个元素的容器对象，常用的比如：list 列表对象、 tuple 元组对象、dict 字典对象、set 集合对象。Python 定义这些类型的变量，语法非常简洁。

举例如下。

使用一对中括号 `[]`，创建一个 list 型变量：

```
1 lst = [1,3,5] # list 变量
```

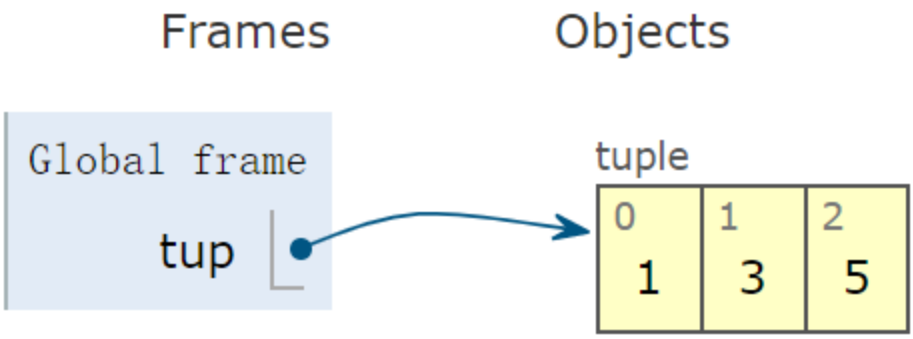
示意图看出，右侧容器为开环的，意味着可以向容器中增加和删除元素：




使用一对括号 `()`，创建一个 tuple 型对象：

```
1 tup = (1,3,5) # tuple 变量
```

示意图看出，右侧容器为闭合的，意味着一旦创建元组后，便不能再向容器中增删元素：




**但需要注意，含单个元素的元组后面必须保留一个逗号，才被解释为元组。**

```
1 tup = (1,) # 必须保留逗号
```

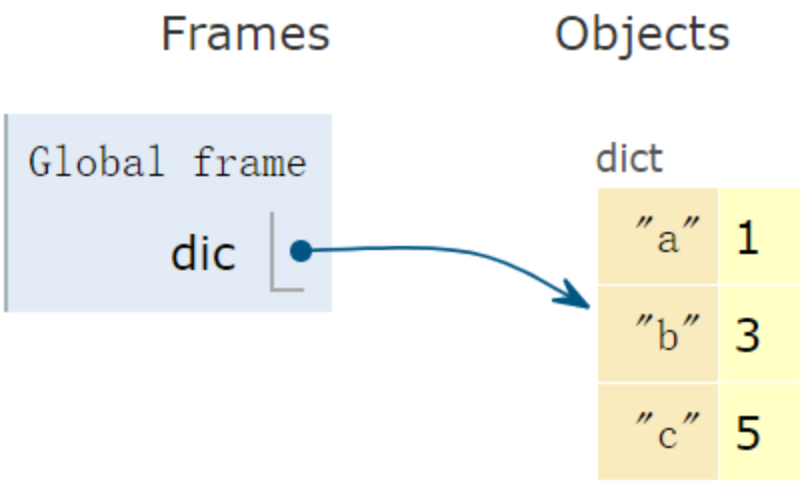
否则会被认为元素本身：

```
1 In [14]: tup=(1)
2     ...: print(type(tup))
3 <class 'int'>
```

使用一对花括号 `{}` 另使用冒号 `:`，创建一个 dict 对象：

```
1 dic = {'a':1, 'b':3, 'c':5} # dict变量
```

字典是一个哈希表，下面的示意图形象的表达出字典的“形”。




仅使用一对花括号 `{}`，创建一个 set 对象：

```
1 s = {1,3,5} # 集合变量
```

Python 的容器类型，list、dict、tuple、set 等能方便地实现强大的功能，下面给出几个案例。

## 1. 去最求平均「去掉最大值和最小值」

在开始写出最后代码之前，我们先把需要的基础知识，写出来。那就是：round。

### 1.1 Python3 round() 函数

**描述**

round() 方法返回浮点数 x 的四舍五入值，准确的说保留值将保留到离上一位更近的一端（四舍六入）。

精度要求高的，不建议使用该函数。

**语法**

以下是 round() 方法的语法:

```
1 round( x [, n]  )
```

**参数**

- x -- 数字表达式。
- n -- 表示从小数点位数，其中 x 需要四舍五入，默认值为 0。

**返回值**

> 返回浮点数 x 的四舍五入值。

**实例**

以下展示了使用 round() 方法的实例：

```
print("round(70.23456) : ", round(70.23456))
print("round(56.659,1) : ", round(56.659, 1))
print("round(80.264, 2) : ", round(80.264, 2))
print("round(100.000056, 3) : ", round(100.000056, 3))
print("round(-100.000056, 3) : ", round(-100.000056, 3))
```

输出结果：

```
round(70.23456) :  70
round(56.659,1) :  56.7
round(80.264, 2) :  80.26
round(100.000056, 3) :  100.0
round(-100.000056, 3) :  -100.0
```

看下官网给的一个例子：

```
>>> round(2.675, 2)
2.67
```

**按我们的想法返回结果应该是 2.68，可结果却是 2.67，为什么？**

> 这跟浮点数的精度有关。我们知道在机器中浮点数不一定能精确表达，因为换算成一串 1 和 0 后可能是无限位数的，机器已经做出了截断处理。那么在机器中保存的2.675这个数字就

> 比实际数字要小那么一点点。这一点点就导致了它离 2.67 要更近一点点，所以保留两位小数时就近似到了 2.67。
>
> 因为该函数对于返回的浮点数并不是按照四舍五入的规则来计算，而会收到计算机表示精度的影响。
> 关于该问题搜索后解释比较清楚的文章地址如下：
> http://www.runoob.com/w3cnote/python-round-func-note.html

接下来，我们来写最终目标代码：去掉列表中的一个最小值和一个最大值后，计算剩余元素的平均值。

```
def score_mean(lst):
    lst.sort()
    lst2 = lst[1:-1]
    return round((sum(lst2) / len(lst2)), 1)


lst = [9.1, 9.0, 8.1, 9.7, 19, 8.2, 8.6, 9.8]
print(score_mean(lst))  # 9.1
```

代码执行过程，动画演示：

## 2. 打印 99 乘法表

打印出如下格式的乘法表：

```
1*1=1
1*2=2   2*2=4
1*3=3   2*3=6   3*3=9
1*4=4   2*4=8   3*4=12  4*4=16
1*5=5   2*5=10  3*5=15  4*5=20  5*5=25
1*6=6   2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7   2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8   2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9   2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*
9=81
```

一共有 10 行，第 i 行的第 j 列等于：`j*i`，其中：

- i 取值范围：`1<=i<=9`
- j 取值范围：`1<=j<=i`

根据“例子分析”的语言描述，转化为如下代码：

```
In [13]: for i in range(1,10):
    ...:     for j in range(1,i+1):
    ...:         print('%d*%d=%d'%(j,i,j*i),end='\t')
    ...:     print()
```

## 3. 样本抽样

使用 sample 抽样，如下例子从 100 个样本中随机抽样 10 个。

```
from random import randint, sample

lst = [randint(0, 50) for _ in range(100)]
```

```
print(lst[:5])  # [38, 19, 11, 3, 6]
lst_sample = sample(lst, 10)
print(lst_sample)  # [33, 40, 35, 49, 24, 15, 48, 29, 37, 24]
```

## 1.3 字符串

注意 Python 中没有像 C++ 表示的字符类型（char），所有的字符或串都被统一为 str 对象。如单个字符 `c` 的类型也为 str。

str 类型会被经常使用，先列举 5 个被高频使用的方法。

- strip 用于去除字符串前后的空格：

```
In [1]: '  I love python\t\n  '.strip()
Out[1]: 'I love python'
```

- replace 用于字符串的替换：

```
In [2]: 'i love python'.replace(' ','_')
Out[2]: 'i_love_python'
```

- join 用于合并字符串：

```
In [3]: '_'.join(['book', 'store','count'])
Out[3]: 'book_store_count'
```

- title 用于单词的首字符大写：

```
In [4]: 'i love python'.title()
Out[4]: 'I Love Python'
```

- find 用于返回匹配字符串的起始位置索引：

```
In [5]: 'i love python'.find('python')
Out[5]: 7
```

举个应用字符串的案例，判断 str1 是否由 str2 旋转而来。

字符串 stringbook 旋转后得到 bookstring，写一段代码验证 str1 是否为 str2 旋转得到。

**转化为判断：**str1 是否为 str2+str2 的子串。

下面函数原型中，注明了每个参数的类型、返回值的类型，增强代码的可读性和可维护性。

```python
def is_rotation(s1: str, s2: str) -> bool:
    if s1 is None or s2 is None:
        return False
    if len(s1) != len(s2):
        return False

    def is_substring(s1: str, s2: str) -> bool:
        return s1 in s2

    return is_substring(s1, s2 + s2)
```

测试函数 is_rotation：

```python
r = is_rotation('stringbook', 'bookstring')
print(r)  # True

r = is_rotation('greatman', 'maneatgr')
print(r)  # False
```

代码执行过程，动画演示：

Frames Objects

字符串的匹配操作除了使用 str 封装的方法外，Python 的 re 正则模块功能更加强大，写法更为简便，广泛适用于爬虫、数据分析等。

下面这个案例实现：密码安全检查，使用正则表达式非常容易实现。

密码安全要求：

- 要求密码为 6 到 20 位；
- 密码只包含英文字母和数字。

```
import re

pat = re.compile(r'\w{6,20}')  # 这是错误的，因为 \w 通配符匹配的是字母，数字和下划线，题目要求不能含有下划线
# 使用最稳的方法：\da-zA-Z 满足“密码只包含英文字母和数字”
# \d匹配数字 0-9
# a-z 匹配所有小写字符；A-Z 匹配所有大写字符
pat = re.compile(r'[\da-zA-Z]{6,20}')
```

选用最保险的 fullmatch 方法，查看是否整个字符串都匹配。

以下测试例子都返回 None，原因都在解释里。

```
pat.fullmatch('qaz12') # 返回 None，长度小于 6
pat.fullmatch('qaz12wsxedcrfvtgb67890942234343434') # None 长度大于
 22
pat.fullmatch('qaz_231') # None 含有下划线
```

下面这个字符串 `n0passw0Rd` 完全符合：

```
In [20]: pat.fullmatch('n0passw0Rd')
Out[20]: <re.Match object; span=(0, 10), match='n0passw0Rd'>
```

## 1.4 自定义类型

Python 使用关键字 class 定制自己的类，self 表示类实例对象本身。

一个自定义类内包括属性、方法，其中有些方法是自带的。

**类（对象）：**

```
class Dog(object):
    pass
```

以上定义一个 Dog 对象，它继承于根类 object，pass 表示没有自定义任何属性和方法。

下面创建一个 Dog 类型的实例：

```
wangwang = Dog()
```

Dog 类现在没有定义任何方法，但是刚才说了，它会有自带的方法，使用 __dir__() 查看这些自带方法：

```
In [26]: wangwang.__dir__()
Out[26]:
['__module__',
 '__dict__',
 '__weakref__',
 '__doc__',
 '__repr__',
 '__hash__',
 '__str__',
 '__getattribute__',
 '__setattr__',
 '__delattr__',
 '__lt__',
 '__le__',
 '__eq__',
 '__ne__',
 '__gt__',
 '__ge__',
 '__init__',
 '__new__',
 '__reduce_ex__',
 '__reduce__',
 '__subclasshook__',
 '__init_subclass__',
 '__format__',
 '__sizeof__',
 '__dir__',
 '__class__']
```

有些地方称以上方法为魔法方法，它们与创建类时自定义个性化行为有关。比如：

- __init__ 方法能定义一个带参数的类；
- __new__ 方法自定义实例化类的行为；
- __getattribute__ 方法自定义读取属性的行为；
- __setattr__ 自定义赋值与修改属性时的行为。

**类的属性：**

```
def __init__(self, name, dtype):
    self.name = name
    self.dtype = dtype
```

通过 __init__，定义 Dog 对象的两个属性：name、dtype。

**类的实例：**

```
wangwang = Dog('wangwang','cute_type')
```

`wangwang` 是 `Dog` 类的实例。

**类的方法：**

```
def shout(self):
    print('I\'m %s, type: %s' % (self.name, self.dtype))
```

注意：

- 自定义方法的第一个参数必须是 self，它指向实例本身，如 Dog 类型的实例 dog；
- 引用属性时，必须前面添加 self，比如 `self.name` 等。

总结以上代码：

```
In [40]: class Dog(object):
    ...:     def __init__(self,name,dtype):
    ...:         self.name=name
    ...:         self.dtype=dtype
    ...:     def shout(self):
    ...:         print('I\'m %s, type: %s' % (self.name, self.dty
pe))

In [41]: wangwang = Dog('wangwang','cute_type')

In [42]: wangwang.name
Out[42]: 'wangwang'

In [43]: wangwang.dtype
Out[43]: 'cute_type'

In [44]: wangwang.shout()
I'm wangwang, type: cute_type
```

看到创建的两个属性和一个方法都被暴露在外面，可被 wangwang 调用。这样的话，这些属性就会被任意修改：

```
In [49]: wangwang.name='wrong_name'

In [50]: wangwang.name
Out[50]: 'wrong_name'
```

如果想避免属性 name 被修改，可以将它变为私有变量。改动方法：属性前加 2 个 _ 后，变为私有属性。如：

```
In [51]: class Dog(object):
    ...:     def __init__(self,name,dtype):
    ...:         self.__name=name
    ...:         self.__dtype=dtype
```

```
5     ...:     def shout(self):
6     ...:         print('I\'m %s, type: %s' % (self.name, self.dtyp
  e))
```

同理，方法前加 2 个 _ 后，方法变为“私有方法”，只能在 Dog 类内被共享使用。

但是这样改动后，属性 name 不能被访问了，也就无法得知 wangwang 的名字叫啥。不过，这个问题有一种简单的解决方法，直接新定义一个方法就行：

```
1 def get_name(self):
2     return self.__name
```

综合代码：

```
1  In [52]: class Dog(object):
2      ...:     def __init__(self,name,dtype):
3      ...:         self.__name=name
4      ...:         self.__dtype=dtype
5      ...:     def shout(self):
6      ...:         print('I\'m %s, type: %s' % (self.name, self.dty
   pe))
7      ...:     def get_name(self):
8      ...:         return self.__name
9      ...:
10
11 In [53]: wangwang = Dog('wangwang','cute_type')
12
13 In [54]: wangwang.get_name()
14 Out[54]: 'wangwang'
```

但是，通过此机制，改变属性的可读性或可写性，怎么看都不太优雅！因为无形中增加一些冗余的方法，如 get_name。

下面，通过另一个例子，解释如何更优雅地改变某个属性为只读或只写。

自定义一个最精简的 Book 类，它继承于系统的根类 object：

```
class Book(object):
    def __init__(self,name,sale):
        self.__name = name
        self.__sale = sale
```

使用 Python 自带的 property 类，就会优雅地将 name 变为只读的。

```
    @property
    def name(self):
        return self.__name
```

使用 @property 装饰后 name 变为属性，意味着 `.name` 就会返回这本书的名字，而不是通过 `.name()` 这种函数调用的方法。这样变为真正的属性后，可读性更好。

```
In [101]: class Book(object):
     ...:     def __init__(self,name,sale):
     ...:         self.__name = name
     ...:         self.__sale = sale
     ...:     @property
     ...:     def name(self):
     ...:         return self.__name

In [102]: a_book = Book('magic_book',100000)

In [103]: a_book.name
Out[103]: 'magic_book'
```

property 是 Python 自带的类，前三个参数都是函数类型。更加详细的讨论放在后面讨论装饰器时再展开。

```
In [104]: help(property)
Help on class property in module builtins:

class property(object)
 |  property(fget=None, fset=None, fdel=None, doc=None)
```

如果使 name 既可读又可写，就再增加一个装饰器 @name.setter。

```
In [105]: class Book(object):
     ...:     def __init__(self,name,sale):
     ...:         self.__name = name
     ...:         self.__sale = sale
     ...:     @property
     ...:     def name(self):
     ...:         return self.__name
     ...:     @name.setter
     ...:     def name(self,new_name):
     ...:         self.__name = new_name

In [106]: a_book = Book('magic_book',100000)

In [107]: a_book.name = 'magic_book_2.0'

In [108]: a_book.name
Out[108]: 'magic_book_2.0'
```

注意这种装饰器写法：name.setter，name 已经被包装为 property 实例，调用实例上的 setter 函数再包装 name 后就会可写。对于 Python 入门者，可以暂时不用太纠结这部分理论，使用 Python 一段时间后，再回过头来自然就会理解。

# 小结

今天学习 Python 的四大基本数据类型。数值型 int、float 等；容器型 list、dict、tuple、set 等；字符型 str 与正则表达式介绍；自定义类的基本语法规则，class、属性和方法等。

**动画对应短视频下载链接：**

Python60 天专栏录制短动画：

> https://pan.baidu.com/s/1MrnrXEnl54XDYYwkZbSqSA
> 提取码：634k

Day2.mp4