

# STAT243 Lecture 8.1 Basic Representations

## 1 位与字节 (Bits and Bytes)

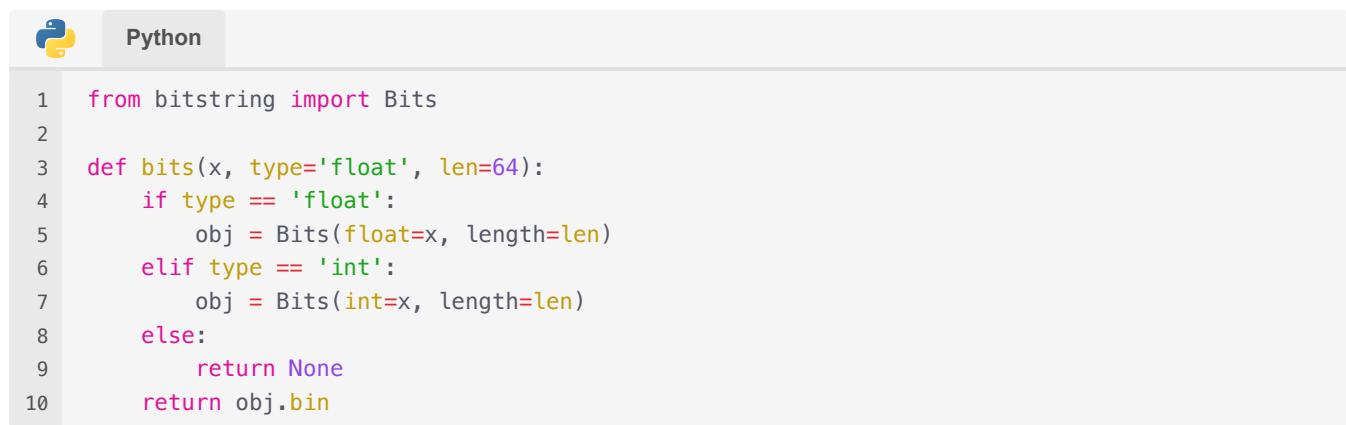
- 一切数据（包括文本、图像、声音）在计算机中最终都以二进制形式存储。
- Bit (位)：最小存储单位，可取 0 或 1（相当于一个开关）。
- Byte (字节)：由 8 位组成。

### 1.1 ASCII 文本编码

- ASCII 使用 1 个字节（8 位）存储一个字符，但实际仅用到 7 位，可表示 128 ( $2^7$ ) 个字符。
- 字节的值范围是 0–255（共 256 个值）。
- 为简洁起见，我们通常用 十六进制 (hexadecimal) 表示字节内容，例如：
  - 3e, a0, ba 等。
- 文件格式 (file format) 只是对文件中字节序列的一种解释方式。

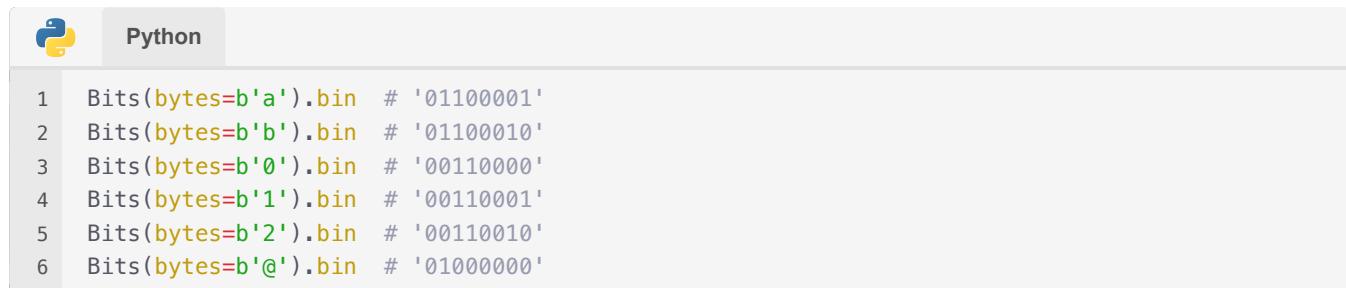
## 2 查看二进制表示 (Binary Representation with `bitstring`)

示例函数：



```
Python
1 from bitstring import Bits
2
3 def bits(x, type='float', len=64):
4     if type == 'float':
5         obj = Bits(float=x, length=len)
6     elif type == 'int':
7         obj = Bits(int=x, length=len)
8     else:
9         return None
10    return obj.bin
```

### 2.1 示例：字符的 ASCII 编码



```
Python
1 Bits(bytes=b'a').bin # '01100001'
2 Bits(bytes=b'b').bin # '01100010'
3 Bits(bytes=b'0').bin # '00110000'
4 Bits(bytes=b'1').bin # '00110001'
5 Bits(bytes=b'2').bin # '00110010'
6 Bits(bytes=b'@').bin # '01000000'
```

说明：

- 'b' 的编码比 'a' 大 1
- '1' 的编码比 '0' 大 1
- 可查阅 ASCII 编码表验证

## 3 整数的二进制存储 (Integer Representation)

- **两字节 (16 bits)** → 可表示 0 到  $2^{16} - 1 = 65535$  (无符号整数)。
  - 若使用一个 bit 作为符号位 (正负号), 则可表示约 -32768 到 32767。

实际中，负数并非以“符号位 + 数字”的形式存储，而是采用 **二进制补码 (two's complement)** 表示，便于计算机执行加减法。

### 3.1 示例：64位整数表示

## 说明：

在补码表示中， $-1$  的二进制为全 1，便于计算（例如  $1 + (-1) = 0$ ）。

## 4 整数溢出 (Integer Overflow)

计算机中的整数集并非对加法封闭。若结果超出可表示范围，将发生 **溢出 (overflow)**。



## Python

```
1 a = np.int32(3423333)
2 a * a # overflow warning
3 # np.int32(-1756921895)
4
5 a = np.int64(3423333)
6 a * a
7 # np.int64(11719208828889)
```

- `int32` 溢出，而 `int64` 不会（因为范围更大）。

进一步：

```
Python

1 a = np.int64(34233332342343)
2 a * a
3 # np.int64(1001093889201452977)    # 溢出结果错误
4
5 a = 34233332342343
6 a * a
7 # 1171921043261307270950729649    # 正确
```

Python 的原生 `int` 类型支持任意精度整数，不会溢出，但代价是使用更多内存。  
例如：

 Python

```
1 import sys
2 a = 34233332342343
3 a * a
4 # 1171921043261307270950729649
5
6 sys.getsizeof(a)      # 32 bytes
7 sys.getsizeof(a * a)  # 36 bytes
```

## | 5 浮点数 (Floating Point Numbers)

- 在 C / NumPy 中常见的实数类型：
  - `float32`: 单精度 (4 字节)
  - `float64`: 双精度 (8 字节)
- GPU 运算中常偏向使用单精度以节省内存与提升速度。

### | 5.1 示例：内存占用

 Python

```
1 import numpy as np, sys
2 x = np.random.normal(size=100000)
3 sys.getsizeof(x)  # float64, ~800 KB
4
5 x = np.array(np.random.normal(size=100000), dtype="float32")
6 sys.getsizeof(x)  # ~400 KB
7
8 x = np.array(np.random.normal(size=100000), dtype="float16")
9 sys.getsizeof(x)  # ~200 KB
```

**规律：**每降低一倍字节长度，内存占用近似减半。

### | 5.2 内存估算公式

若使用双精度浮点数：

$$\text{Memory (MB)} = \frac{N \times 8}{10^6}$$

其中  $N$  为元素个数。

| 注意：有时计算机使用 **MiB (Mebibyte)**，即  $1, \text{MiB} = 2^{20} \text{ bytes}$ 。

## | 6 NumPy 数值信息查询 (Machine Information)

### | 6.1 使用 `iinfo` 函数查询数值范围

NumPy 提供辅助函数查看不同数据类型的数值范围：



Python

```
1 np.iinfo(np.int32)
2 # iinfo(min=-2147483648, max=2147483647, dtype=int32)
3
4 np.iinfo(np.int64)
5 # iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

以 32-bit (4-byte) integer 为例, 最大整数为  $2147483647 = 2^{31} - 1$ , 如果考虑正数负数和零的话, 总共可以表示  $2 \cdot 2^{31} = 2^{32}$  个数

## 6.2 示例：32 位整数边界



Python

```
1 np.binary_repr(2147483647, width=32)
2 # '01111111111111111111111111111111' # 最大正数
3
4 np.binary_repr(-2147483648, width=32)
5 # '10000000000000000000000000000000' # 最小负数
```

尝试超过范围会报错：



Python

```
1 np.int32(2147483648)
2 # OverflowError: Python integer 2147483648 out of bounds for int32
```

## 7 总结

类型	字节数	位数	可表示范围	示例类型
int16	2	16	$-32768 \sim 32767$	np.int16
int32	4	32	$-2,147,483,648 \sim 2,147,483,647$	np.int32
int64	8	64	$-9.22 \times 10^{18} \sim 9.22 \times 10^{18}$	np.int64
float32	4	32	单精度浮点	np.float32
float64	8	64	双精度浮点	np.float64

## 要点回顾

- 计算机底层数据均为二进制位。
- 整数使用补码表示以简化算术操作。
- 有限位数导致溢出；Python 原生整数可避免此问题。
- 浮点类型影响精度与内存使用。
- NumPy 的 `iinfo` 可查看整数范围, `sys.getsizeof` 可检测内存使用。