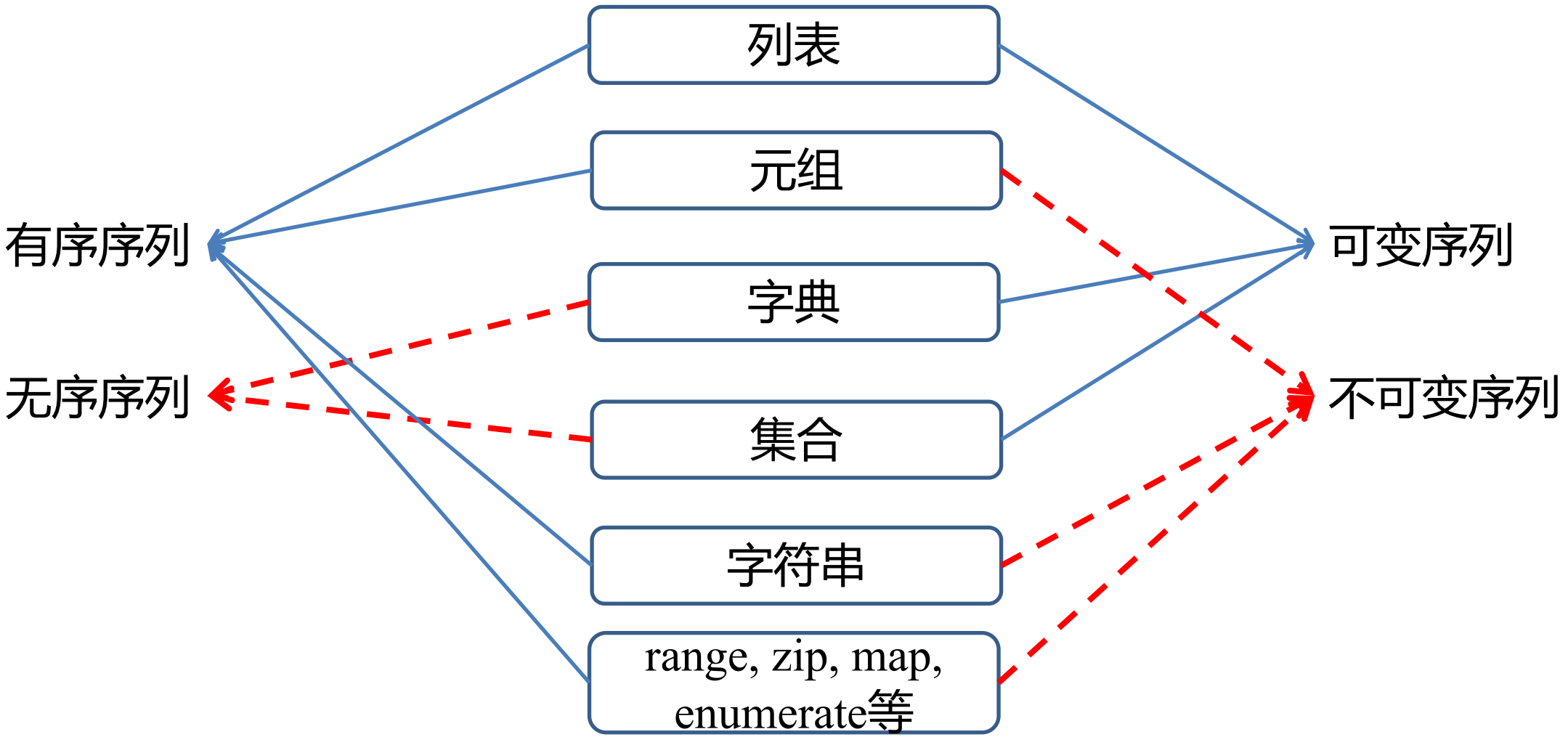
東南大學 SOUTHEAST UNIVERSITY 人工智能学院

Python编程

授课教师：王洪松

邮箱：hongsongwang@seu.edu.cn

Python序列结构



不可变数据类型

- Python中，不可变数据类型包括：整数、浮点数、复数、字符串、元组等。
- 不可变数据类型意味着一旦对象被创建，其内容就不能被改变。如果你尝试“修改”一个不可变对象，Python实际上会创建一个新的对象来存储新的值，并返回这个新对象的引用。

```
a = 5          # 创建一个整数对象，并让变量a指向它
b = a          # 让变量b也指向同一个整数对象

print(a is b) # 输出True，因为a和b指向同一个对象

c = a + 1      # 创建一个新的整数对象（值为6），并让变量c指向它
print(a is c) # 输出False，因为a和c指向不同的对象
```

可变数据类型

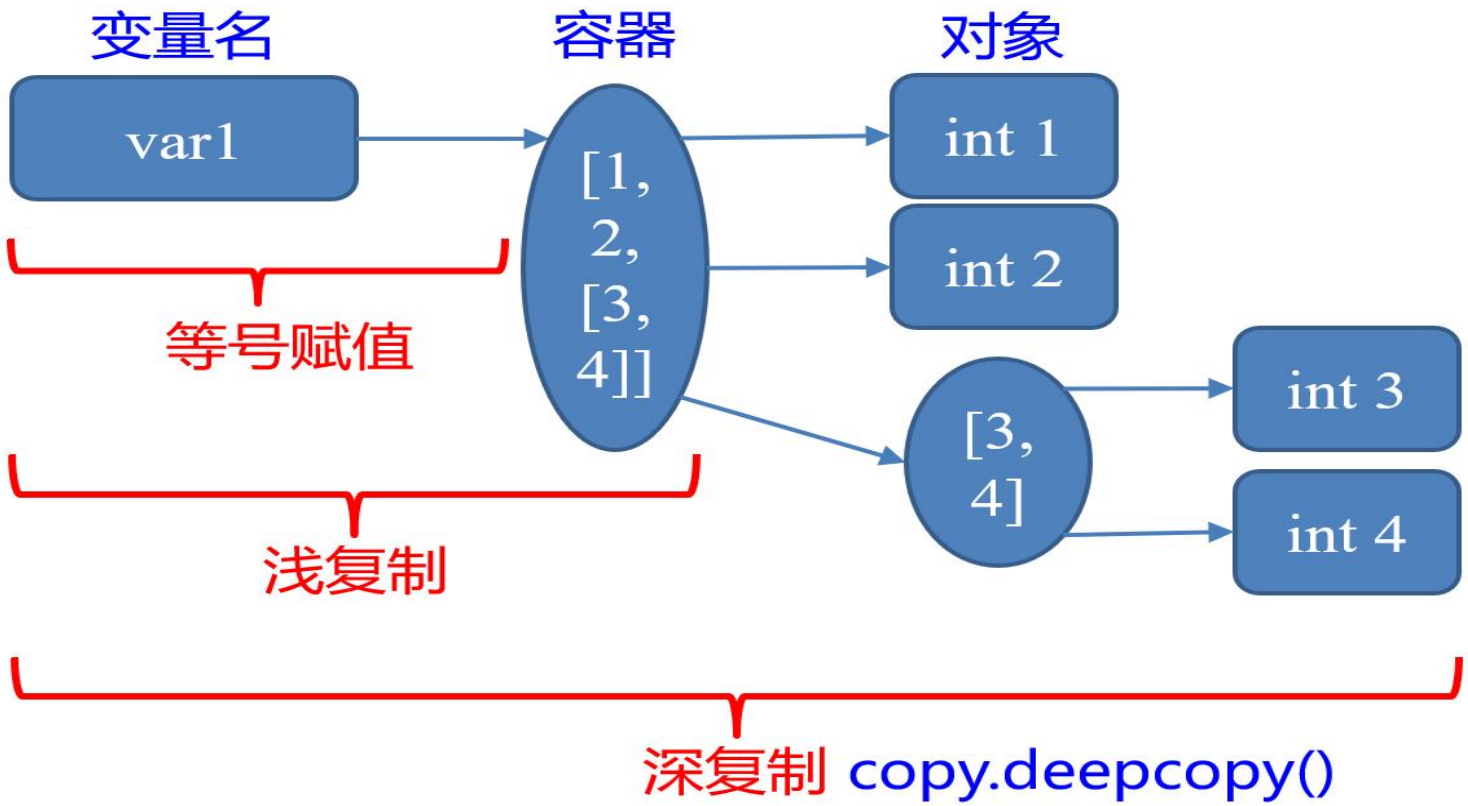
- 可变数据类型指的是那些允许在对象创建后修改其内容的数据类型。与不可变数据类型（如整数、浮点数、字符串、元组等）相比，可变数据类型提供了更大的灵活性。
- 可变数据类型包括：列表、字典、集合、自定义对象。

```
a = [1, 2, 3]  # 创建一个列表
b = a
print(a is b)  # True

a.append(4)    # 追加一个元素
print(a is b)  # True
```

浅复制与深复制

- Python中赋值其实就是对象的引用（指针）
- 浅复制：复制了外部的对象本身，内部的元素都只是复制了一个引用
- 深复制：把对象复制一遍，也复制该对象中引用的其他对象



Python序列结构
元组、字典等

Python序列结构：元组

- 元组是有序、不可变序列
- 元组创建
  - tuple()方法 #相当于list()方法于列表
  - tuple()冻结列表，list()融化元组
- 元组不支持添加元素
  - 没有append()、extend()和insert()方法
- 元组不支持删除元素
  - 没有remove()和pop()方法

Python序列结构：元组

- 元组的访问速度比列表更快
  - 如果定义了一系列常量值，而所需的仅是对它进行遍历，则使用元组更有优势
- 元组对数据进行“写保护”，使得代码更安全
- 元组因其不可变，可作为字典的“键”，也可作为集合的元素
- 元组中只包含一个元素时，需要在元素后面添加逗号来消除歧义

```
#coding=utf-8
list_1 = [1, 2, 3, 4, 5]
tuple_1 = tuple(list_1)
# tuple_1 = (1, 2, 3, 4, 5)
tuple_2 = (6,)
```

Python序列结构：元组

- 理解元组的不可变性质
  - 元组是不可变类型，这意味着一旦创建后，其地址和大小都不能被改变
  - 对于元组中的不可变元素，如数字、字符串等，不能修改其值
  - 如果元组中包含可变类型的元素，例如列表，虽然不能修改元组的地址，但是可以修改列表内部的值，因为列表是可变类型

```
a = (1, 2, 3, [4, 5])
# 修改元组中的列表元素
a[3].append(6)
print(a)  # 输出：(1, 2, 3, [4, 5, 6])

a[3] += [9]
TypeError: 'tuple' object does not support item assignment
a[3] = a[3] + [10]
TypeError: 'tuple' object does not support item assignment
```

Python序列结构：元组

- 理解元组的不可变性质

```
var1 = (1,2,3)
var2 = copy.copy(var1) #浅复制

var3 = [1,2,3]
var4 = copy.copy(var3) #浅复制

print(id(var1),id(var2),id(var3),id(var4))


#id(var1) = id(var2)
#id(var3) ≠ id(var4)
```

Python序列结构：元组

```
# 创建一个包含不可变元素的元组
original_tuple = (1, 2, 3)
shallow_copy_tuple = original_tuple[:]  # 对元组进行浅复制
print(id(original_tuple))  # 输出原始元组的内存地址
print(id(shallow_copy_tuple))  # 通常也会输出相同的内存地址，元组是不可变的

# 创建一个包含可变元素的元组
mutable_tuple = ([1, 2], [3, 4])
shallow_copy_mutable_tuple = mutable_tuple[:] # 对元组进行浅复制
# 检查原始元组和浅复制元组的地址
print(id(mutable_tuple))  # 输出原始元组的内存地址
print(id(shallow_copy_mutable_tuple))  # 输出一个不同的内存地址，因为虽然元组本身是不可变的，但它包含的列表是可变的
print(id(mutable_tuple[0]))  # 输出原始元组中第一个列表的内存地址
print(id(shallow_copy_mutable_tuple[0]))  # 输出相同的内存地址
```

Python序列结构：zip方法

- zip() 函数用于将若干可迭代对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的可迭代对象

```
a = [1, 2, 3]
b = [4, 5, 6]
c = [7, 8, 9, 10]
zipped = zip(a, b)
print(list(zipped)) # [(1, 4), (2, 5), (3, 6)]
print(list(zip(a, c))) # [(1, 7), (2, 8), (3, 9)]
print(list(zip(a, b, c))) # [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
a1, a2, a3 = zip(*zip(a, b, c)) # 与zip相反，zip(*)可理解为解压，返回二维矩阵式
# a1 = (1, 2, 3)
# a2 = (4, 5, 6)
# a3 = (7, 8, 9)
```

Python序列结构：字典

- 字典是无序、可变的序列
  - 每个元素的键和值用冒号分隔，所有元素放在一对大括号中
- 字典中的键可为任意不可变的数据类型
  - 如整数、实数、复数、字符串、元组等
- globals()返回含当前作用域内所有全局变量和值的字典
- locals()返回含当前作用域内所有局部变量和值的字典

Python序列结构：字典

- 为什么要设计字典？
  - 键值对映射：在实际问题中，需要通过某个唯一标识去查找对应的信息
  - 高效的查找和更新
    - 查找：平均时间复杂度 O(1)
    - 插入/更新：平均时间复杂度 O(1)
    - 删除：平均时间复杂度 O(1)
  - 表达能力强：字典可以用来表示现实世界中复杂的数据结构

Python序列结构：字典

- 字典的创建
  - 使用“=”将一个字典赋值给一个变量
  - 使用dict()方法利用已有数据创建字典

```
1    dict_1 = {'University': 'SEU', 'Title': 'Dr.'} # = 直接赋值
2    keys = [1, 2, 3, 4]
3    values = ('a', 'b', 'c', 'd')
4    dict_2 = dict(zip(keys, values))
5    # dict_2 = {1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

Python序列结构：字典

- 字典的创建
  - 使用dict()方法根据给定的键、值创建字典
  - 以给定内容为键，创建值为空的字典

```
6   dict_3 = dict(name='Xiaoming', age=16, gender='male')
7   #dict_3 = {'name': 'Xiaoming', 'age': 16, 'gender': 'male'}
8   dict_4 = dict.fromkeys(['name', 'age', 'gender'])
9   #dict_4 =  {'name': None, 'age': None, 'gender': None}
```

Python序列结构：字典

- 字典元素的读取
  - 以键为下标可以读取字典元素，若键不存在则抛出异常
  - 使用字典对象的get方法获取指定键对应的值，并可以在键不存在的时候返回指定值

```
1    dict_1 = dict(name='Xiaoming', age=16, gender='male')
2    print(dict_1['age']) # 16
3    print(dict_1.get('age')) # 16
4    print(dict_1.get('school')) # None
5    print(dict_1.get('school', 'No.1 High School')) # No.1 High School
```

Python序列结构：字典

- 字典的遍历
  - items()方法返回字典的键、值对
  - keys()方法返回字典的键
  - values()方法返回字典的值

```
1    dict_1 = dict(name='Xiaoming', age=16, gender='male')
2    for i in dict_1: # 默认输出Key
3        print(i)
4    for i in dict_1.items(): # 输出元组格式的键值对
5        print(i)
```

```
name
age
gender
('name', 'Xiaoming')
('age', 16)
('gender', 'male')
```

Python序列结构：字典

- 字典元素的增加和修改
  - 当以指定键为下标为字典元素赋值时
    - 若键存在，则修改该键对应的值
    - 若键不存在，则增加一个键、值对
  - 使用update()方法将另一个字典的键、值对添加至当前字典对象

```
1  dict_1 = dict(name='Xiaoming', age=16, gender='male')
2  dict_2 = {1: 'a', 2: 'b'}
3  dict_3 = dict.fromkeys(['name', 'age', 'gender'])
4  dict_1.update(dict_2)
5  # dict_1 = {'name': 'Xiaoming', 'age': 16, 'gender': 'male', 1: 'a', 2: 'b'}
6  dict_1.update(dict_3)
7  # dict_1 = {'name': None, 'age': None, 'gender': None, 1: 'a', 2: 'b'}
```

Python序列结构：字典

- 字典元素的增加和修改
  - 使用del命令删除字典中指定键的元素
  - 使用字典对象的clear()方法删除字典中所有元素
  - 使用字典对象的pop('key')方法删除并返回指定键的元素
  - 使用字典对象的popitem()方法删除并返回字典中的一个随机元素

Python序列结构：集合

- 集合是无序、可变序列，使用大括号界定，元素不可重复
- 集合中只能包含数字、字符串、元素等不可变类型（可哈希）
- 集合的创建
  - 使用“=”将集合赋值给变量
  - 使用set()函数将其他序列转变为集合

```
1    set_1 = {1, 3, 5}
2    set_2 = set(range(5)) # set_2 = {0, 1, 2, 3, 4}
```

Python序列结构：集合

- 集合的创建与删除
  - 使用del命令删除整个集合
  - 使用remove(元素)方法删除指定元素
  - 使用pop()方法删除其中一个随机元素
  - 使用clear()方法清空集合

Python序列结构：集合

- 集合支持交集、并集、差集等运算

```
1  set_1 = set(range(1, 5))
2  set_2 = {3, 4, 5, 6, 7}
3  print(set_1 | set_2) # 并集 {1, 2, 3, 4, 5, 6, 7}
4  print(set_1 & set_2) # 交集 {3, 4}
5  print(set_1 - set_2) # 差集 {1, 2}
6  print(set_1 ^ set_2) # 对称差集 {1, 2, 5, 6, 7}
```

- 集合判断子集

```
1  set_1 = {1, 2, 3}
2  set_2 = {1, 2, 4}
3  set_3 = {1, 2, 3, 4}
4  print(set_1.issubset(set_3)) # 判断子集 True
5  print(set_2 < set_3) # 判断子集 True
```

Python序列结构：字符串

- 字符串属于**不可变**数据类型，除了支持序列通用方法（包括切片操作）以外，还支持特有的字符串操作方法

```
1  a = 'Southeast'
2  print(a[0]) # 支持下标访问，S
3  print(a[::-1]) # 支持切片操作，tsaehtuoS
4  for i in a: #支持循环访问其中字符
5      print(i)
```

Python序列结构：字符串

- Python的缓存机制会影响数据安全
  - 不同解释器实际使用缓存机制会有所区别
    - 数字类型

```
>>> a = 1
>>> b = 1
>>> a == b
True
>>> a is b
True
>>> a = 257
>>> b = 257
>>> a == b
True
>>> a is b
False
```

IDLE/命令行

```
a = 1
b = 1
print('a,b 都是1时:', a is b)
a = 257
b = 257
print('a,b 都是257时:', a is b)
```

```
a,b 都是1时: True
a,b 都是257时: True
```

Spyder/PyCharm

Python序列结构：字符串

- Python的缓存机制会影响数据安全
  - 不同解释器实际使用缓存机制会有所区别
  - 字符串类型

```
>>> a = 'abc'
>>> b = 'abc'
>>> a is b
True
```

```
>>> a = '东'
>>> b = '东'
>>> a is b
False
```

Python序列结构：字符串

- 字符串简单？
  - 编码是第一道关
- 编码：信息从一种形式转换为另一种形式的过程
  - 数字编码：2进制、10进制...
  - 字符编码：名词眼花缭乱！
    - ASCII码
    - ANSI
    - GB2312
    - GBK
    - Unicode
    - utf-8

Python序列结构：字符串

- 字符编码基本知识
  - 存储单位
    - 位（bit）：计算机存储信息的最小单位
    - 字节（Byte）：是一种计量单位，8个二进制位组成1个字节
    - 1个字符占用若干个字节
  - 字符集
    - ASCII .....，Unicode
  - 编码方式
    - UTF-8
    - UTF-16

Python序列结构：字符串

- 字符集
  - ASCII码
    - American Standard Code for Information Interchange，美国标准信息交换码
    - 每个ASCII字符占用1个Byte（8bits）
    - 共有256位字符或符号
    - 10个数字，大小写英文字母，常用符号，控制信号占用了0-127的字符
    - 128-255的字符是扩展字符

Python序列结构：字符串

- 字符集
  - GB2312
    - ASCII码一共才256个字符，而且还被用了，汉字怎么办?
    - 中国的办法：保留127以前的字符，以后的换成双字节的汉字，组合出7000多个汉字
    - 半角/全角
  - GBK
  - GB18030
  - DBCS编码方案（Double Byte Character Set）
  - 香港、台湾...

  - **以上统称ANSI编码，** American National Standards Institute

Python序列结构：字符串

- 字符集
  - Unicode
    - 统一码、万国码
    - ISO规定必须用16位来表示一个字符
    - 对于ASCII里的那些“半角”字符， Unicode包持其原编码不变，只是将其长度由原来的8位扩展为16位
    - 其他文化和语言的字符则全部重新统一编码
  - Unicode的问题
    - 两个字节表示英文字母过于浪费空间
      - UTF（UCS Transfer Format）诞生了

Python序列结构：字符串

- 编码方式
  - UTF-8 每次8个位传输数据
  - UTF-16 每次16个位传输数据
  - 从Unicode到UTF时并不是直接的对应，而是要过一些算法和规则来转换

- UTF-8 非常惊艳，用1-4个字节表示一个字符
  - 对于单个字节的字符，第一位设为 0，后面的 7 位对应这个字符的 Unicode 码点
  - 对于需要使用 N 个字节来表示的字符（N > 1），第一个字节的前 N 位都设为 1，第 N + 1 位设为0，剩余的 N - 1 个字节的前两位都设位 10，剩下的二进制位则使用这个字符的 Unicode 码点来填充

Python序列结构：字符串

- Python解释器一般默认ASCII编码，如包含中文，为防止乱码，往往需要在编码开头重新声明编码类型

```
# -*- coding: <encoding name> -*- : # -*- coding: utf-8 -*-
# coding=<encoding name>：  # coding=utf-8
# coding：<encoding name>：  # coding: utf-8
```

- coding不可省略，如 # utf-8 起不到声明编码的作用！

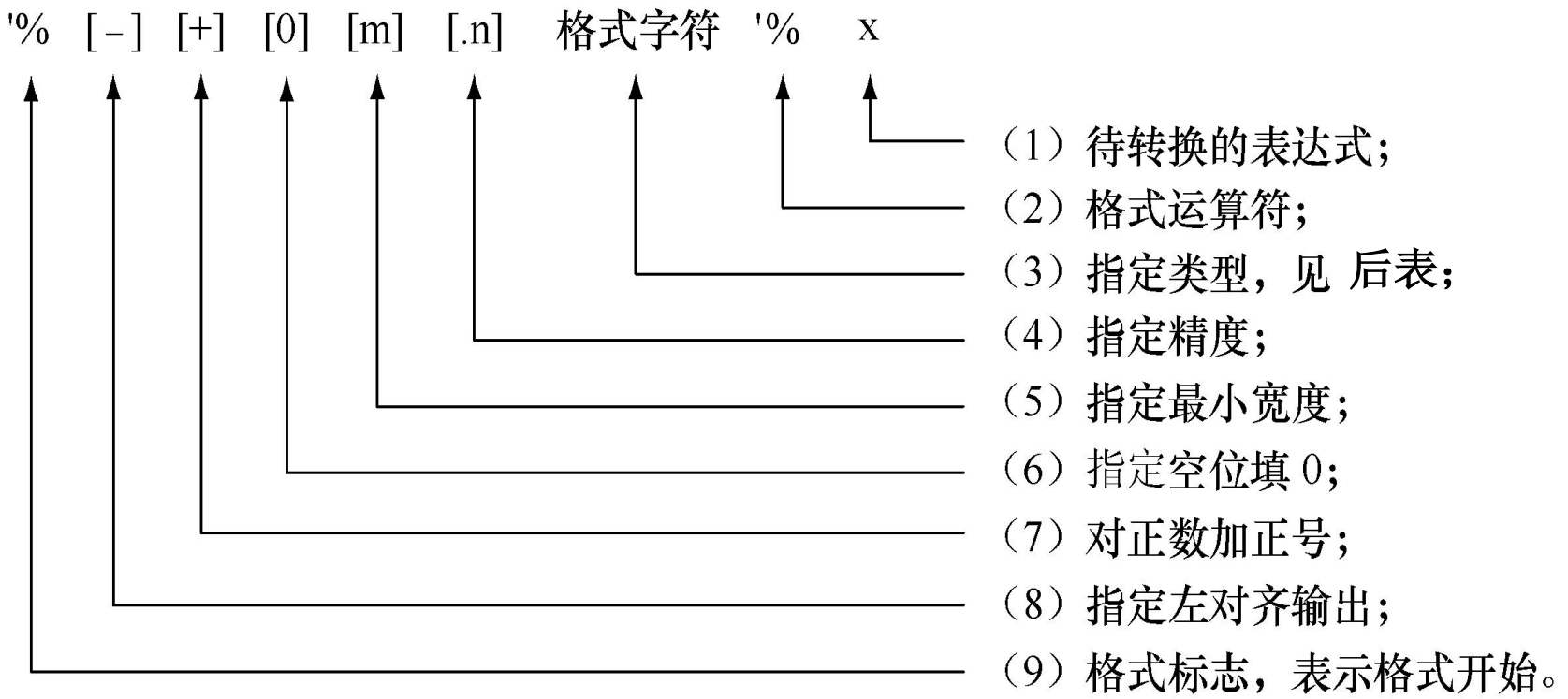
4.1 字符串

- Python中的编码示例

```
print('东南'.encode('utf-8')) # b'\xe4\xb8\x9c\xe5\x8d\x97'
print('东南'.encode('gbk')) # b'\xb6\xab\xc4\xcf'
print('东南'.encode('cp936'))# b'\xb6\xab\xc4\xcf'
print('东南'.encode('utf-8').decode('gbk')) # 涓滃崡
print('东南'.encode('gbk').decode('utf-8')) # UnicodeDecodeError
```

Python序列结构：字符串

- 字符串的格式化
  - “%格式化” 的基本语法格式



Python序列结构：字符串

- 字符串的格式化

格式字符	说明
%s	字符串 (采用str()的显示)
%r	字符串 (采用repr()的显示)
%c	单个字符(chr()函数转换)
%b	二进制整数(只能用于字符串format方法和format函数)
%d和%i	十进制整数(带符号)
%u	十进制整数(不带符号)
%o	八进制整数
%x和%X	十六进制整数, 区别在于输出时大小写
%e和%E	指数 (基底写为e或E)
%f和%F	浮点数
%g和%G	指数(e/E)或浮点数 (根据显示长度)
%%	输出字符%

Python序列结构：字符串

- Python3.6以后，引入字符串的 f-string 格式化

```
1    name = 'Xiaoming'
2    age = 16
3    print('%s is %s' % (name, age)) # Xiaoming is 16
4    print('{} is {}'.format(name, age)) # Xiaoming is 16
5    print(f'{name} is {age}') # Xiaoming is 16
```

Python序列结构：字符串

- 字符串常用方法
  - find()和rfind()方法分别用来查找一个字符串在另一个字符串指定范围（默认是整个字符串）中首次和最后一次出现的位置，如果不存在则返回-1

```
fruits = "apple, peach,banana, peach, pear"
print(fruits.find("peach")) # 7
print(fruits.find("peach", 9)) # 21
print(fruits.find("peach", 9, 21)) # -1
print(fruits.rfind("peach")) # 21
print(fruits.rfind("peach", 9)) # 21
print(fruits.rfind("peach", 9, 21)) # -1
```

Python序列结构：字符串

- 字符串常用方法
  - index()和rindex()方法用来返回一个字符串在另一个字符串指定范围中首次和最后一次出现的位置，如果不存在则抛出异常
  - count()方法用来返回一个字符串在另一个字符串中出现的次数

```
fruits = "apple, peach,banana, peach, pear"
print(fruits.index("peach")) # 7
print(fruits.index("p")) # 1
print(fruits.index("ppp")) # ValueError
print(fruits.count("p")) # 5
print(fruits.count("pp")) # 1
print(fruits.count("ppp")) # 0
```

Python序列结构：字符串

- 字符串常用方法
  - split()和rsplit()方法分别用来以指定字符为分隔符，将字符串左端和右端开始将其分割成多个字符串，并返回包含分割结果的列表
  - partition()和rpartition()用来以指定字符串为分隔符将原字符串分割为3部分，即分隔符前的字符串、分隔符字符串、分隔符后的字符串，如果指定的分隔符不在原字符串中，则返回原字符串和两个空字符串。

```
fruits = "apple, peach,banana, peach, pear"
print(fruits.split()) # ['apple,', 'peach,banana,', 'peach,', 'pear']
print(fruits.split(',')) # ['apple', ' peach', 'banana', ' peach', ' pear']
print(fruits.partition(',')) # ('apple', ',', ' peach,banana, peach, pear')
print(fruits.rpartition(',')) # ('apple, peach,banana, peach', ',', ' pear')
print(fruits.rpartition('banana')) # ('apple, peach,', 'banana', ', peach, pear')
```

Python序列结构：字符串

- 字符串常用方法
  - 对于split()和rsplit()方法，如果不指定分隔符，则字符串中的任何空白符号（包括空格、换行符、制表符等等）都将被认为是分隔符，返回包含最终分割结果的列表
  - split()和rsplit()方法还允许指定最大分割次数
    - str.split("char",num)
      - char：表示分隔标识符
      - num：表示分隔最大次数，为空表示分隔所有

Python序列结构：字符串

- 字符串常用操作
  - 字符串联接join()
    - 'sep'.join(seq)
      - sep：分隔符，可以为空
      - seq：要连接的元素序列、字符串、元组等
      - 以sep作为分隔符，将seq所有的元素合并成一个新的字符串

```
a = '东南大学'
b = '-'
print(b.join(a)) # 东-南-大-学
```

  - lower(), upper(), capitalize(), title(), swapcase()这几个方法分别用来将字符串转换为小写、大写字符串、将字符串首字母变为大写、将每个单词的首字母变为大写以及大小写互换
  - strip(), rstrip(), lstrip()这几个方法分别用来删除两端、右端或左端的空格或连续的指定字符

Python序列结构：字符串

- 字符串常用操作
  - eval(expression[, globals[, locals]])
    - expression -- 表达式
    - globals -- 变量作用域，全局命名空间，如果被提供，则必须是一个字典对象
    - locals -- 变量作用域，局部命名空间，如果被提供，可以是任何映射对象
  - 计算字符串中有效的表达式
  - 将字符串转成相应的对象

注意： eval() 函数执行的代码具有潜在的安全风险。如果使用不受信任的字符串作为表达式，则可能导致代码注入漏洞，因此，应谨慎使用 eval() 函数，并确保仅执行可信任的字符串表达式。

```
# 执行简单的数学表达式
result = eval("2 + 3 * 4")
print(result)  # 输出: 14

# 执行变量引用
x = 10
result = eval("x + 5")
print(result)  # 输出: 15

# 在指定命名空间中执行表达式
namespace = {'a': 2, 'b': 3}
result = eval("a + b", namespace)
print(result)  # 输出: 5
```

作业题

- 用3种方法创建一个字典并初始化
- 给定key获取字典中的value
  - 如果key不在字典里，返回一个默认值
  - 至少用2种方法实现以上操作
- 用至少3种方法合并两个字典，比较它们的计算时间
- 列表的列表和字典互相转换
  - 把字典转换为列表的列表，即列表中每个元素也是一个列表，该列表的第一个元素为key，其它元素为value
  - 将列表的列表转换为字典
  - 以上均至少用2种方法实现

作业题

- 简单字符串操作
  - 字符串反转并输出
  - 输出奇数位上的子字符串
  - 输出字符串的后k位字符串
  - 将字符串中小写字母变成大写字母
  - 用多种方法判断字符串是否以“.jpg”结尾
  - 删除字符串中的重复元素
  - 输出包含两个字符串中的公共字符的字串

作业题

- 统计元素出现的次数
  - 定义一个列表
  - 用random.sample(.)函数从列表中采样N个样本
  - 删除所定义的列表，统计采样生成的列表中有哪些元素并计算它们出现的次数，结果以字典的形式输出

- 用字典存储树（数据结构）
  - 定义一个字典存储右图的数据结构树
  - key为节点的编号，value为长度为2的列表，2个元素分别为左右子树