计算导论

-
- tips：
  - 转义字符在考察范围内，算在 format string 里面。
  - 考试分成 30 题选择题（75 分）以及 2 题程序小题（25 分）。答案请写在试卷首页的表格以及方框中。考试中的选择题大部分是问 print 出的信息，但少数会问“正确的”或者“错误的”语句的数量，请大家特别看清楚。
  - 各位同学，目前计划期末考试大致上分为约 30+道选择题与两道小程序题。小程序题的代码行数一般不超过 10 行。小程序的一个往年样题是小作业题里面计算 hamming code 的那道题。大家可以考虑一下如果不用 bin()函数，该怎么写（这个可能会超过 10 行）。为缓解批改压力，会有专门的答题卡填选择题的答案以及专门的方框填程序题的答案。写在其他地方的答案不计。超出方框的程序代码原则上也不计。
  - 考试范围内
  - 选择题是单选题。
  - 程序小题每个最多写 10 行
  - 考试能带 50 页 A4，这已经很宽松了。去年是只能带 1 页 A4
  - 请大家尽量带 25 页双面打印的资料，50 页还是比较多的。
  - match 高级匹配和 decorator 的用法仅限于 ppt 以及 ipynb 里面的例子
- L7 递归的函数编写
- 其他：
  - 语句顶格写，开头不能任意加空格
  - 建议每行 79 个字符以内

基本类型

整数、浮点数、复数、字符串、逻辑真值的表示

整数 integer

- 默认**十进制**
- Binary(**二进制**): 0b (0B) 96 = 0b1100000 or 0B1100000
- Octal(**八进制**): 0o (0O) 96 = 0o140 or 0O140
- Hexadecimal(**16 进制**): 0x (0X) 96 = 0x60 or 0X60.
  - 10,11,…,15 are ‘ABCDEF’ or ‘abcdef’
  - 8 = 0x8, 9 = 0x9, 10= 0xA or 0xa, …, 15 = 0xF or 0xf.
- **下划线**来提高可读性 (Python 3.6)
  - 123,456,789 可以使用 123_456_789 来表示
  - 0xFFF_FFF_FAA, 123.456_789
- **完整存储**，没有位数限制

浮点数 float

- 近似保存，浮点数不准确，精度有限制
  - 范围 2.2250738585072014e-308 到 1.7976931348623157e+308

- 舍入误差 Rounding error
- print(123.00)：123.0
- print(1e4):10000.0
- print(.1):0.1
- print(.1+.1+.1）：0.3000000000000004
- x =13.94999999999999999 print(x):13.95

- **复数 complex number**
  - a+bj，b=1 时 1 不能忽略
  - print(1+2j):(1+2j)
  - 使用 z.real, z.imag 访问 z 的实部和虚部，但不能修改因为 immutable
  - complex(real, imag)构造虚数，参数均可选，默认值为 0
    - z2 = complex(3, 4) print(z2) 输出(3+4j)
- **科学计数法**
  - aeb :a × 10^b 1e3=10^3 1e+3=10^+3 1.2e-5=1.2*10^-5
  - e 的前后不能加空格
- **逻辑真值**
  - True is 1, False is 0.大小写敏感，True*12=12
  - **False**：[], (), {}, False, 0, None
  - **True**：非零数字和非空字符串
  - **逻辑操作符** not and or 优先级非和或
- **逻辑表达式**
  - x 被翻译为 bool（x）
  - **bool** 是一种特殊形式的 int
    - print（None == False）：Flase
    - print（type（True））：class ‘bool’
  - 可能不直接返回 True 和 False，而是返回相等的值
    - 1+2+3>6 or 3, return 3, bool(3)==True
    - 0 and “abc”>”ABC”, return 0, bool(0) ==False
  - **短路求值 Short-circuit evaluation**
    - True or x, False and y 中 x 和 y 不会被判断
  - x < y <= z 等于 x < y and y <= z，不过前者 y 只计算一次，如果 x<y 为 False，z 不会被计算
  -

```python
def f(n):
    if n == 1:
        return True
    else:
        return False


# 等价于
def f(n):
    return n == 1
```

- **基本类型转换函数： int(), float(), str(), bool(), bin()**
  - **int()**
    - 浮点数，字符串到整数，直接截断浮点数的小数部分，只能转换整数字符串
    - int() 不一定总是下取整，例如 int(4.9999999999999999) = 5
    - print(int("3.14")):ValueError
    - print(int(float("3.14"))可以，返回 3
  - **float()**
    - 字符串，整数到浮点数
  - **bin()**
    - a=bin(10) 输出 0b1010
      - 去掉 0b 用 string 的 slicing 就行了
    - 转换为二进制。
    - x.bit_length()计算二进制值长度
    - x.bit_count()计算 1 的个数
- **算术运算符：+, -, *, /, //, %, **, +=, -=, *= /=, //=, %=, **=**
  - 和数学中一样，运算符有优先级，** 高于 *，/； *, / 高于+，-括号()可以改变运算顺序
  - **中不能加空格
  - **除法/**：运算结果为 float
  - **整除//**：a, b 都为 int，结果为 int; a, b 有一个为 float，结果为 float
  - **余数%**
    - 判断两个数同余，不能用 a == v, 要用 (a-b)%n == 0
    - 在计算大整数的余数的时候，用同余运算来处理，计算量更小
  - 常见错误：x^0.5 写成 x＊＊ 1/2
  - 连等及+=，同时赋值都采用之前的值，不会在计算中改变
- **逻辑运算符：and, or, not, is, in, ==, !=, <=, >=, <, >**
  - >, <, ==, >=, <=, !=, in, not in, is, is not 优先级相同
  - **is** 判断 id 相同吗，==判断值相同吗
  - 5.0==5， 4.9999999999999999999=5

- **位运算：|, &, ^, >>, << (取反操作~x 不考)**
  - >>右移运算符将一个数的所有位向右移动指定的位数，左侧的空位填充 0。等价于除以二的 n 次方
    - 8 >> 1 将二进制数 1000 向右移动 1 位，得到 0100，等于十进制数 4。
  - <<左移运算符将一个数的所有位向左移动指定的位数，右侧的空位填充 0。等价于乘以二的 n 次方

&	Bitwise AND	x & y	Bitwise AND operator: Returns 1 if both the bits are 1 else 0.
|	Bitwise OR	x | y	Returns 1 if either of the bit is 1 else 0
~	Bitwise NOT	~x	Returns one's complement of the number ($0 \rightarrow 1$, $1 \rightarrow 0$)
^	Bitwise XOR	x ^ y	Returns 1 if one of the bit is 1 and other is 0 else returns 0
>>	Bitwise right shift	x>>	Shifts the bits of the number to the right and fills 0 on voids left as a result. Similar effect as of dividing the number with some power of two
<<	Bitwise left shift	x<<	Shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two

  - 位运算符 优先级 比 算术运算符 低（3 + 4 << 1 等价于 (3 + 4) << 1）
  - n&1 判断是不是偶数，因为比较的是一个二进制 00001，前面一定全都是 0，比较最后一个。n>>1 相当于除以二

```
n = 127
while n&1:
    n >>= 1
print(n)
```

- **range 的写法**
  - range(a,b,d):(从 a 到 b 以 d 为公差的等差数列， 不包括 b) d 默认为 1，可以省略.a, b, d 必须都为整数
  - x = range(100)，type(x)为<class 'range'>

变元与赋值语句

- **变元**
  - 数字，字母，下划线，用字母或者下划线开头，文件名也是
  - 大小写敏感，惯例不使用大写字母
  - 有 35 个关键字：['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
  - 变量 type 由 python 决定
- **指向关系**
- **重新赋值 (a = ...)**
  - 改变 a 的地址
  - 同时赋值<var>, <var>, ..., <var> = <expr>, <expr>, ..., <expr>
- **修改赋值 (a[i] = ..., a.x = ..., ...)**

- **immutable vs. mutable**
  - += 的区别 a = a+b 重新赋值，a+=b 对于 mutable 对象是修改
  - mutable 对象不复用之前已有具有相同内容的对象
- 赋值语句中的表达式如果是单个变元 a，那么直接返回 a 指向的对象而不作 copy。
- 赋值语句表达式中的每一次运算原则上都会创建新的对象。immutable 对象有可能会共享之前创建的对象。

条件分支

- == 判断两个对象的内容是否相等，is 判断两个对象是否是同一个（即是否在同一个地址）。
- “a” or False 返回什么值之类的不考。
- **基本语法: if, elif, else, match**
  - 如果用多个 if 而没有用 elif，那么 else 只判断最后一个 if 的 else
  - if 语句中至少有一行语句，可以用 pass 跳过
- **a if b else c 形式**
  - return a if b else c 等价于 if b: return a else: return c
- 条件不为逻辑表达式的判定（例如 if “a”：等等）

循环

- **for 循环**
  - for 循环的集合可以是 list, dict, tuple, set, str, range, etc.（iterable）
  - 在 for x in xxx: do_sth()中 do_sth()对 x 的修改并不改变 x 的遍历顺序
    - 用 while 手工操作
    - 用集合 collection_a.copy()来处理
    - 用 for x in reversed(lst):
    - 列表推导
  - 会无限循环

```
x = ['ab', 'cd']
for i in x:
    x.append(i.upper())
print(x)
```

- **while 循环**
  - while x: #bool(x)==True
- **for … else 循环**
- **while … else 循环**
  - 当循环正常结束的时候，也既没有被 break，运行 else 语句
  - 如果没有 break 语句，else 会永远执行
- **break 和 continue**
  - break:跳出当前循环
  - continue: 跳出当次循环迭代，继续循环

- **for 循环中使用 list, dict，set 的注意点**
  - dict 在循环中修改大小，会报错。
  - 如果 list, dict 在循环中不改变大小，也就是说只改变值是可以的，可以直接修改 index 以及 key 上的 value
  - 如果需要修改容器的大小，一般用.copy()在副本上遍历，或者改用 while 循环。

函数

- **语法以及调用方式**
  - def function_name(param1, param2, ...):return ......
  - 一个函数调用由 函数名+参数 决定，参数不同、函数名相同也是不同的调用
  - 被调用后才会运行，运行后函数内部的代码和数据被清除，是一次性的（销毁变元不一定销毁他们指向的对象）
- **参数类型： positional, keyword, arbitrary positional, arbitrary keyword**
  - 参数数量要一致，对齐，参数传递赋值语句 x1,x2,x3=y1,y2,y3
  - 函数调用时候输入的参数实际上是输入对象的地址
  - default arguments 默认参数
  - **positional** 位置参数
  - **keyword** 关键字参数
  - **arbitrary positional 任意位置参数**
    - *args 输入参数，作为**元组**传递

```
def f(*args):
    for x in args:
        print(x)


f(1, 2, 3)
f(3, 4)
f(7)
```

  - **arbitrary keyword 任意关键字参数**
    - **kwargs 接受关键词参数，作为**字典**传递，kwargs 等不是关键词，是惯例，可以修改

```
def test_kwargs(**kwargs):
    for x in kwargs:
        print(kwargs[x], end=' ')

    print()


test_kwargs(x=1)
test_kwargs(x=1, y=2)
test_kwargs(x=1, y=2, z=3)
test_kwargs(x=1, y=2, z=3, u=4)
```

  - 特定位置参数

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      ----------     ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                  - Keyword only
         -- Positional only
```

```
def f(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)

f(10, 20, 30, d=40, e=50, f=60)
# f(10, b=20, c=30, d=40, e=50, f=60)   # b cannot be a keyword argument
# f(10, 20, 30, 40, 50, f=60)           # e must be a keyword argument
```

  - 规则

- 默认参数在非默认参数之后

```
def example(a, b=2):  # 正确
    print(a, b)

def example(b=2, a):  # 错误，默认参数不能在非默认参数之前
    print(a, b)
```

- 关键字参数在位置参数之后

```
def example(a, b):
    print(a, b)

example(1, b=2)  # 正确，关键字参数 b 放在位置参数 1 之后
example(a=1, 2)  # 错误，位置参数 2 不能放在关键字参数 a 之后
```

- **缺省参数以及摆放位置的限制**
  - 缺省参数必须出现在所有位置参数之后
  - 要按顺序摆放，或者通过 keyword 传递
  - 小心可变对象作为缺省参数的默认值，可以将其设为 None 来避免
- **return**
  - return 立即终止函数执行，如果没有执行 return 语句则返回 None
    - None 与“”，0 False 都不一样，是 NoneType
  - 多个返回值是 tuple 形式
- **局部变量和全局变量**互不干涉
  - **global** 如果要在函数内修改函数外部某个变量，可以在前面加 global 关键字
  - 可以在函数内部直接使用外部的变量，如果不修改这个变量（如果重新赋值 a=2 不报错，如果 a+=1 报错）
  - 函数内部和外部变元可以有有限度的重叠，只要不在函数内部又用外部变元又把它当成内部变元就行（上面不行下面可以）

```
a = 100
def addone():
    b = a + 1
    a = b+1
    print(b)
addone()

def addone(a):
    b = a + 1
    print(b)
addone(50)
```

- **传入的对象实质是对象的地址**
  - 重新赋值不影响外部对象
  - 对象函数调用、修改赋值或者对象成员修改会影响外部对象的内容（如果传入参数是 mutable 的话）
- **类型提示**

```python
def add(x: float, y: float) -> float:
    return x + y
```

- **decorator**
- 函数定义内部也可以嵌套定义函数，也可以返回内部嵌套定义的函数。
- **递归机制**
  - 一个终止状态：简单情况直接计算
  - 一个化简状态：复杂情况转化为简单的情况
  - stack：
    - push 堆叠起来，A1,A2,...An 被 push 到 stack 里
    - pop 一个个 remove，执行 An，pop up An-1,...,A1
    - 整个过程叫 stack（FIFO：First in last out）
- **对象函数调用语法（例如 lst.count()）**
- **基本的 lambda 语法**
  - lambda 表达式（也称为匿名函数）是在 Python 中创建小型匿名函数的一种方式
  - lambda 参数列表 : 表达式
    - lambda：关键字，表示这是一个匿名函数。
    - 参数列表：用逗号分隔的参数列表，类似于常规函数的参数。
    - 表达式：单个表达式，匿名函数计算并返回其值。
    - square = lambda x: x**2
    - print(square(5)) # 输出: 25
    - add = lambda x, y: x + y
    - print(add(3, 5)) # 输出: 8
- **晚绑定 late binding**
  - 函数中使用的外部变量由调用的时候的值决定，而不是定义时的值决定

```python
def pow_x():
    plist = []
    for i in range(5):
        def powa(x):
            return x**i
        plist.append(powa)#这是把powa函数加入到plist中
    return plist
for i in range(5):
    print(pow_x()[i](2))
```

```
16
16
16
16
16
```

  - funcs = [lambda x: x + i for i in range(5)]
  - print([f(10) for f in funcs]) # 输出: [14, 14, 14, 14, 14]
  - 在这个例子中，所有的 lambda 函数捕获了 i 的最终值（4）。这是因为在 lambda 函数定义时，i 的值并没有被确定，而是在 lambda 函数实际调用时才确定。这也体现了晚绑定的概念。
- 缩进推荐使用空格，不能和 tab 混用

- 后面的函数会覆盖前面的函数

内置函数

print(str, end, sep)

- end 表示结尾符，默认是回车
- sep 表示分隔符，默认是空格
- print()表示单空一行，因为默认 end="\n"
- print(123),print('123')效果一样
- print(str1,str2,...,strn,end="SDDW\n",sep="aa")

eval()

- 注意 eval 可以处理很广泛的表达式，例如 eval("[1,2,3]"）返回列表[1,2,3]
- x=eval("1+2+3") print(x):6
- eval 把一个 string 翻译为一个 python 表达式并计算他的值
- eval 可以将一个字符串形式的 list，转换为 list：lst = eval（“[1,2,3,[1,2,3]]”）
- program = input('Enter a program:')　　eval(program) #用户输入[print(item) for item in [1, 2, 3]]，那么输出 1,2,3
- eval 只适用于表达式，不适用于赋值语句 Error: eval（“a=1”）

exec()

- 执行创造的程序。返回 None。
- exec("ax=-1.234")　exec("print(ax)")输出-1.234

type()

- 返回类型

id()

- 返回地址

input()

- 返回一个字符串!
- 回车键（\n）终止输入
- <variable> = input(<prompt>)
- input 一次输入多个对象
  - x,y,z = tuple(input().split(','))
  - x,y,z = float(x), float(y), float(z)

len()

- 返回容器里面元素的多少

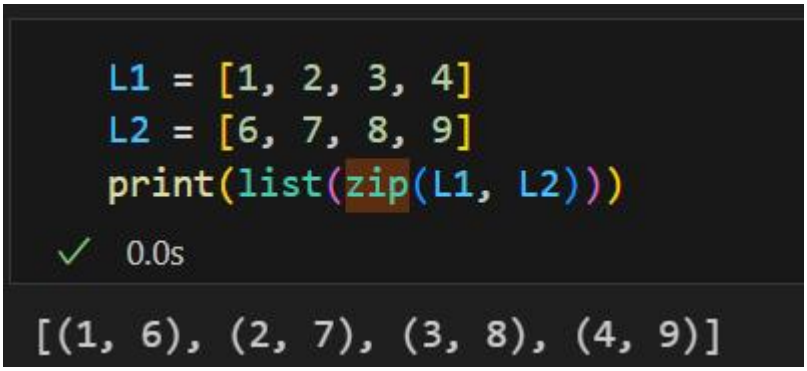
sum()

- 计算可迭代对象所有元素的总和，元素必须是数字类型（整数、浮点数、复数）
- sum(iterable, start)
  - iterable：这是一个包含数字的可迭代对象（如列表、元组等）。
  - start（可选）：这是一个指定的初始值，默认值为 0。计算总和时，它会被加到可迭代对象的元素之和中。

- 如果可迭代对象为空，且未指定 start 参数，sum() 返回 0。

- **map()**
  - map() 函数会将一个函数应用于可迭代对象（如列表、元组等）中的每个元素，返回一个新的迭代器。map(function, iterable)
  - def square(x):
  - return x * x
  - numbers = [1, 2, 3, 4, 5]
  - squared_numbers = map(square, numbers)
  - print(list(squared_numbers)) # 输出: [1, 4, 9, 16, 25]
- **filter()**
  - filter() 函数会将一个函数应用于可迭代对象中的每个元素，并根据函数的返回值（True 或 False）过滤元素，返回一个新的迭代器。filter(function, iterable)，True 返回，False 不要
  - def is_even(x):
  - return x % 2 == 0
  - numbers = [1, 2, 3, 4, 5]
  - even_numbers = filter(is_even, numbers)
  - print(list(even_numbers)) # 输出: [2, 4]
- **reduce()**
  - reduce() 是 Python 的 functools 模块中的一个函数，用于对可迭代对象中的元素进行累积计算。它可以应用一个函数（通常是二元函数）于可迭代对象的前两个元素，然后将结果与下一个元素继续进行计算，直到处理完所有元素并返回最终结果。
    - from functools import reduce
    - reduce(function, iterable[, initializer])
    - function：一个二元函数，用于将可迭代对象中的两个元素进行计算，并返回一个结果。
    - iterable：一个可迭代对象，如列表、元组等。
    - initializer（可选）：一个初始值，将作为计算的第一个参数与可迭代对象的第一个元素进行计算。
  - from functools import reduce
  - numbers = [1, 2, 3, 4, 5]
  - total = reduce(lambda x, y: x + y, numbers)
  - print(total) # 输出: 15
- **isinstance()**
  - x = isinstance(object, class)如果是这个类返回 True，不是返回 False
  - class：类或或者包含类的元组。
- **issubclass()**
  - x = issubclass(class, classinfo)如果 class 是 classinfo 的子类（或是其子类之一），则返回 True。否则，返回 False
  - classinfo：可以是单个类或包含多个类的元组。
- **reversed()**

- reversed(seq)反向遍历序列，返回反向迭代器
  - my_list = [1, 2, 3, 4, 5]
  - reversed_list = list(reversed(my_list))
  - print(reversed_list) # 输出: [5, 4, 3, 2, 1]
- **zip()**
  - 将 n 个 list(tuple, str 等等)按元素合并为一个新 list: **每个元素都是 n 维的**



- 变元类型如果和基本函数相同，那么声明出来的变元就会覆盖原来函数，那么原来函数就不能用了

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

List (Mutable)

- **列表的表示**
- **构造函数 list()**
  - list("str")
  - list(range(a,b,n)
  - list()创造空集合
- **加法与乘法**
  - **a+b** 聚合两个列表
    - list1 = [1, 2, 3]
    - list2 = [4, 5, 6]
    - result = list1 + list2
    - print(result) # 输出: [1, 2, 3, 4, 5, 6]
  - **a*n** 将一个列表重复 n 次

    - list1 = [1, 2, 3]
    - result = list1 * 3
    - print(result) # 输出: [1, 2, 3, 1, 2, 3, 1, 2, 3]
- **index 访问与修改**（a[index] = …, b = a[index], …）
    - a[i]超出长度会有 error
    - 负值是倒数
    - list 的修改不改变 list 的 id，只改变内部元素的 id
- **slicing 访问与修改**（a[start:stop:step] = …，b = a[start:stop:step]）
    - step 可以为负数，a[::-1] 返回 a 的逆，step=1 时可以忽略
    - 修改前后的类型应该一致[]=[]
    - 可以通过修改为空列表删除列表元素 a[2,4]=[]
    - 可以通过将空列表修改为列表增加元素 a[2,2]=[1,2,3]
    - id(a[:])与 id(a)不同
    - slice 可以越界，超过范围自动处理报错，非法写法直接返回空列表
        - a = [1,2,3,4] a[3,-4,1]=[], a[:10]=[1,2,3,4]
    - list[::]可以，但是 dict 和 set 不可以，没有切片操作因为无序
- **slicing 缺省值**（a[:]，a[1:], a[:stop]，a[::-1]）
- **in**
    - in 在列表中返回 True
    - not in 在列表中返回 Flase
- **.insert(), .index(), .remove(), .append(), .sort(), .extend(), .count(), .clear()**
    - **list.insert(i,x)**
        - 在 i 个位置插入 x 的值，返回 None，如果值超过列表长度在最后面增加元素
    - **list.append(x)**
        - 在最后增加值直接修改原列表，返回 None
            - a = [0, 1, 2]
            - a.append(a)
            - print(a[3]==a, a[3][3]==a)输出 True，True
    - **list.extend()**
        - 把 interable（list，tuple，string，etc）的所有元素加到 list 的末尾，返回 None

```
4   lst1 = [1, 2]
5   lst2 = [3, 4]
6
7   lst1.extend(lst2)
8
9   print(lst1)
10
11  lst1 = [1, 2]
12  lst2 = [3, 4]
13
14  lst1.append(lst2)
```

```
[1, 2, 3, 4]
[1, 2, [3, 4]]
```

- 不能直接 extend dict，可以 extend dict.values()
  - dict1 = {'a': [1, 2], 'b': [3, 4]}
  - dict2 = {'a': [5, 6], 'b': [7, 8]}
  - for key in dict1:
  - if key in dict2:
  - dict1[key].extend(dict2[key])
  - print(dict1) 输出: {'a': [1, 2, 5, 6], 'b': [3, 4, 7, 8]}

- **list.sort()**
  - 直接修改原列表，返回 None，从大到小排序，元素必须是同种类型
  - 参数 reverse=True 让函数降序排列，默认 False
  - 注意，和 sorted 不同，sorted 是一个函数 b=sorted(a)，创建一个新对象而不是修改，用于任何可迭代对象
- **list.index(x, start, end)**
  - 返回第一个值为 x 的序号，start，end 进行 slice，编号依然是原列表的序号，如果没有 x 的值 error
- **list.count(x)**
  - 返回 list 中 x 的个数，没有其它参数
- **list.clear()**
  - 清除所有元素，返回 None
- **list.pop(index=-1)**
  - 默认删除最后一个元素，并获取它的值
- **list.remove(value)**
  - 删除第一个是这个值的元素，返回 None
- **list.reverse()**
  - 倒转 list，返回 None
- **del lst[index], del lst[start:stop:step]**
  - del a[1:4]通过 index 或者 slice 删除元素

```
a = [10, 20, 30, 40, 50]
del a[1:4]
print(a)  # 输出: [10, 50]
```

  - del a[2], a[3]表示 del a[2] del a[3]，顺序执行故删除的是下标 2 和 4 的元素
  - 尽可能不要循环动态删除一个数据结构，可能出 bug
    - 列表在内存中是顺序排列的，删除后要重新恢复速度慢
    - 标记为不存在的元素例如-1 或者 None
    - 或者用一个新的列表保存结果
- **enumerate()**
  - lst = ['a', 'b', 'c', 'd']
  - for offset, item in enumerate(lst):
  - print(f"Index: {offset}, Value: {item}")

- **(nested) list comprehension 列表推导式**

```
# 示例：生成一个包含平方数的列表
squares = [x**2 for x in range(10)]
print(squares)  # 输出: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

  - 单重循环 [expression for target in iterable if conditon]
  - 多重循环 [expression for target1 in iterable1 if condition1 for target2 in iterable2 if condition2 .. for targetN in iterableN if conditionN]
  - 创建新的 list
    - for
    - [0]*n
    - list(range()),list(set),list(str)
    - list comprehension 清晰可读正确，更快
  - 有 deepcopy 的用处

```
list = [[[1]] for _ in range(3)]
list[0][0][0] = 2
print(id(list[0][0]), id(list[1][0]), id(list[2][0]))
print(list)
```

✓ 0.0s

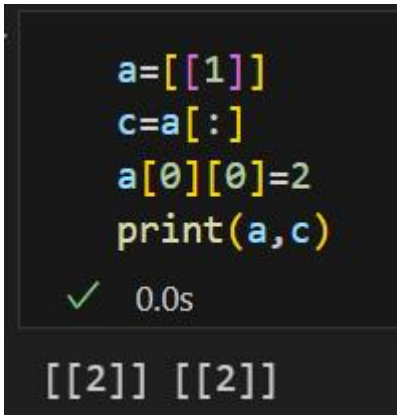
```
2396190861248 2396216187008 2396216196480
[[[2]], [[1]], [[1]]]
```

- **shallow copy & deep copy**
  - **.copy()**会改变第一层的地址，若 b=a 则不改变地址，copy 后 list 的 id 改变，其中元素的 id 不变
    - 如果没有嵌套列表的话没事
  - **copy.deepcopy()**会把嵌套列表所有的地址改变，除了最后一个没有嵌套列表的数据地址不改变
  - *n 操作不作 deepcopy，地址相同

```
a = [1]*3
a[1] = -1
print(a)

a=[[1]]*3
a[1] = -1
print(a)

a=[[1]]*3
a[1][0] = -1
print(a)

a = [[1]*3]*3
a[1][1] = 3.14
print(a)
```

```
[1, -1, 1]
[[1], -1, [1]]
[[-1], [-1], [-1]]
[[1, 3.14, 1], [1, 3.14, 1], [1, 3.14, 1]]
```

  - slicing 类似于 shallow copy(不拷贝里面的对象)
  - b = list(a)，b = a[:]，b = a.copy()，是 shallow copy。



- 列表推导式在每次迭代时都会重新计算推导式中的表达式。 例如[[1] for _ in range(3)] 相当于对[1]作了 deepcopy。

- lst = lst + lst1 和 lst += lst1 前者改变地址重新赋值后者不改变地址
- 系统为了节约空间，有时候会共用不可修改对象，可修改对象一定不会共用
- list 中的元素可以是不同类型的
-

len
Returns an int type specifying number of elements in the collection.

min
Returns the smallest item from a collection.

max
Returns the largest item in an iterable or the largest of two or more arguments.

cmp
Compares two objects and returns an integer according to the outcome.

sum
Returns a total of the items contained in the iterable object.

sorted
Returns a sorted list from the iterable.

reversed
Returns a reverse iterator over a sequence.

all
Returns a Boolean value that indicates whether the collection contains only values that evaluate to True.

any
Returns a Boolean value that indicates whether the collection contains any values that evaluate to True.

enumerate
Returns an enumerate object.

zip
Returns a list of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables.

Tuple (Immutable)

- immutable counterpart of list 元组是不可变列表
- **tuple 表示方法**
  - （）可以被省略 tup = 1, 2, 3
  - 创造单元素 tuple: tup=(1,)或者 1,否则(1)是 int
    - print(type(([1,2,3]))): print(type(([1,2,3])))
    - print(type(([1,2,3],))): <class 'tuple'>
- **构造函数 tuple()**
  - 从列表，字符串，字典（key），元组创建元组

    - 可以通过 a=()创建空 tuple，[1,]是表示[1]不是[(1,)]
- tuple 里面有列表，里面的列表是可以修改的，也就是说 tuple 里面的元素可以是 mutable 的
- tuple 中+会创建一个新列表 tup+=（2,3），而 list 会修改原来列表 lst+=[4]
- 可以嵌套((a, b), c) = ((1, 2), 3)
- **同时赋值** x1, x2, x3, x4 = a, b, c, d

```
x, y, z, w = 1, "2", 3.0, [4j]
a = x, y, z, w
print(type(a))
a1, a2, a3, a4 = a
print(a1, a2, a3, a4)
```

```
<class 'tuple'>
1 2 3.0 [4j]
```

    - 函数中同时 return 多个值用的是 tuple
    - 可以使用 a=b=c=1 赋值，但不推荐
- **index 访问**
    - tuple[index]
- **slicing**
    - tuple[start:stop:step]
- **in**
    - 使用 in 关键字来检查某个值是否存在于元组中
- **.index(x)**
    - 返回 x 的位置，如果不存在发生 error
- **.count(x)**
    - 返回 x 的出现个数
- **加法，乘法**
    - + 将两个元组合并成一个新的元组
    - * 将元组中的元素重复指定次数，生成一个新的元组。
- Python 里面没有 tuple comprehension。

Dict (Mutable)

- **Dict 的表示**
    - 字典使用花括号 {} 表示，其中的键值对用冒号 : 分隔，键值对之间用逗号 , 分隔。
- **Dict 的构造方法（包括 dict.fromkeys()）**
    - **dict()**
        - **关键字参数创建 dict(**kwarg)**
            - d1 = dict(a=1, b=2, c=3)
        - **映射创建 dict(mapping, **kwarg)：**
            - d2=dict({'a': 1, 'b': 2, 'c': 3})
            - d3=dict({'a': 1, 'b': 2}, c=3, d=4)
        - **可迭代对象创建 dict(iterable, **kwarg)**
            - d4 = dict([('a', 1), ('b', 2), ('c', 3)

        - d5 = dict([('a', 1), ('b', 2)], c=3, d=4)
    - **dt = dict({})**
  - **zip 创建**
    - my_dict = dict(zip("xyz", [1, 2, 3])) 输出: {'x': 1, 'y': 2, 'z': 3}
  - **dict.fromkeys()**
    - dict.fromkeys(iterable, value=None)
    - iterable: 一个包含键的可迭代对象。
    - value: 每个键的初始值（默认为 None）
      - keys = ['a', 'b', 'c']
      - newdict = dict.fromkeys(keys, 0)
      - print(newdict) # 输出: {'a': 0, 'b': 0, 'c': 0}
- Dict 的 key 必须 **hashable**（mutable 的 list 和 set 不能作为 key，大部分 immutable 的对象是 hashable 的，tuple 如果里面有 list 那就不 hashable）
- **通过 key 进行访问、修改与增加(dt[key])**
  - 如果 dict[x]，x 不是 key，会 error，不能通过 index 访问字典.3.6+版本后有顺序
  - 通过 key 重新赋值或者增加元素

```
a = {"1":1, "2":2}
a["2"] = -1 # update
a["3"] = 3.14 #add a new one
print(a)
```

- in 和 not in 判断是不是在字典里
- **.keys(), .values(), .items()**
  - **dict.keys()**返回 dict_keys(['test', 1.2, 3, 5])，type 是<class 'dict_keys'>，要自己转成 list：list(a.keys())
  - **dict.values()**返回 dict_values([1, 3, 'hello', [1, 2, 3]])，type 是<class 'dict_values'>
  - **dict.items()**返回 dict_items([('test', 1), (1.2, 3), (3, 'hello'), (5, [1, 2, 3])])，type 是<class 'dict_items'>
- **{**a, **b}, a | b, a.update(b)**
  - **{**a, **b}**融合两个 dict，key 相同以后面的值为准

```
1  a = {"test":1, 1.2:3}
2  b = {2:1, 1.2:-1}
3  c = {** a, ** b}
4  print(c)
```

```
{'test': 1, 1.2: -1, 2: 1}
```

  - **a.update(b)**返回 None，相同的 key 用 b 中的值更新 a 中的值
    - a.update([1,2])
  - **a|b** 合并两个 dict，b 的值取代 a 的值，与**相同
- **get()和 setdefault()**
  - get(a,b)获取 a 的值并返回，如果没有 a，返回 b 设定的 default value，b 默认为 None

    - setdefault(a,b)获取 a 的值并返回，如果不存在，返回 b 设定的 default value，并且在 dict 里面增加 a：b
- **删除元素**
    - **del dict[key]**: 通过 key 删除元素，也可以删除整个字典
    - **dict.clear()**: 创建一个空字典
    - **dict.pop()**: 通过 key 删除元素，返回 value，至少需要一个参数
    - **dict.popitem()**: 删除最后一个 item，返回最后一个键值对
- **多行表达式**
    - **显式**：使用\去把 statement 划分成多行，\后面不能有空格
    - **隐式**：使用(), [], {}、
- **loop**
    - **打印 key**
        - for x in dict: print(x)
    - **打印 value**
        - for x in dict: print(dict[x])
        - for x in dict.values(): print(x)
    - **打印 item**
        - for x in dict.items(): print(x, y)
- 在迭代过程中增加或者删除字典元素是语法错误，可以使用 dict.copy()

```python
n = 5
dt = {x: x for x in range(n)}

for x in dt.copy():
    if dt[x] % 2 == 0:
        del dt[x]

print(dt)
```

```
{1: 1, 3: 3}
```

- **dict comprehension**
    - dt = {x:x for x in range(10)}
- **构造函数是 shallow copy**
    - dict.copy()会改变地址
- dict 的 key 不可以重复，value 可以重复
- value 可以是一个函数，当然 key 也可以是函数只要不可变

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


def sub(x, y):
    return x - y


tool = {"add": add, "sub": sub}
print(tool["add"](1, 2), tool["sub"](3, 3))
```

```
3 0
```

- 一些用法

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns all the tuples for each key value pair
keys()	Returns all the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns all the values in the dictionary

- # Set (Mutable)
  - ## Set 表示方法
    - 无 index
  - ## Set 构造函数
    - {}创造一个空字典而不是空集，set()可以创造空集
    - set()参数为 List, tuple: all the elements；Dict: all the keywords；String: all the character
    - 只能输入一个参数，set(1,2,3)不行
  - Set 里面的元素必须 hashable，([2,])不行
    - set 对象里面的对象必须也是 hashable 的
  - ## in
  - ## 集合运算：a | b, a & b, a ^ b，a-b
    - **a | b 并集**
    - **a & b 交集**
    - **a ^ b 对称差**
    - **a-b 差集**
  - ## .add(), .remove()
    - **a.add(6)**, 返回 None
    - **a.remove(6)**返回 None，若没有则引发异常，而 **a.discard(6)**不会
  - ## <=, <, >, >=
    - A<=B: 返回 True 如果 A 是 B 的子集
    - A<B：返回 True 如果 A 是 B 的真子集
  - ## set comprehension
    - st = {x**2 for x in range(10) if x < 5}
  - **构造函数是 shallow copy**
  - **List, Dict, Set, Tuple 在不打破 Dict/Set 里面的 key 必须 hashable 的情况下可以任意嵌套。**
  - **List, Set,Tuple 的构造函数接受任意 Iterable 对象，其中 Dict 默认的是.keys()。**
  - **内置的构造函数一定会返回一个新对象。**
- # String (Immutable)
  - ## 创建

- “…”和’ …’, 和 “ “ “…” ” ” 和. str()
- 单双引号相同，但不可混用，与最近的匹配
- **index 访问**
  - str[i]
- **slicing**
  - str[start:stop:step]
- **加法和乘法**
  - str1+str2 返回他们的合并
  - str*n 重复 n 次 str，如果 n 小于等于 0 返回空字符串
- **.find(), .split(), .join(), .replace(), .index(), .count(), .upper(), .lower()**
  - **split(seperator)**
    - seperator 可选，如果没有参数，会依照所有空格、换行、tab 进行分割，其中如果有连续的空格、换行、tab 会合并在一起作为分隔符。s.split(" ")只会以空格作为分隔符，并且连续的空格会分隔出空字符串（即连续空格之间认为是空字符串）。
  - **str.join(list)**

```
myTuple = ("John", "Peter", "Vicky")
x = "#".join(myTuple)
print(x)
```

```
John#Peter#Vicky
John:))Peter:))Vicky
```

    - 连接一个 list，tuple，set 或者 dict 的 key，或者字符串，也即所有 iterable 的对象


```
myTuple = ("John", "Peter", "Vicky")
x = "#".join(myTuple)
print(x)
```

```
John#Peter#Vicky
John:))Peter:))Vicky
```


    - text = "ABCDE"
    - result = "-".join(text)
    - print(result) # 输出: "A-B-C-D-E"
  - **str.find(sub, start, end)**
    - 返回第一个 sub 在切片中的位置，start 和 end 可选，如果没找到返回-1

```
fruit = 'banana'
print(fruit.find('a'))
print(fruit.find('na'))
print(fruit.find('na', 3))
print(fruit.find('na', 1, 3))
```

```
1
2
4
-1
```

  - **str.count(sub, start, end)**
    - 返回 sub 在切片中的个数
  - **str.isdigit()**
    - 如果 str 里全是数字并且至少有一个字符，返回 True，否则 False
  - **str.upper(a), str.lower()**
    - 返回 a 的 copy，全大写或全小写
  - **str.strip()**
    - 去除 str 两端的所有空白字符
  - **str.rstrip(), str.lstrip():**

    - 去除右边/左边所有空白字符（包括空格、制表符、换行符等）
  - **str.replace(old, new, count)**
    - 返回 str 的 copy，把 old 替换为 new，次数为 count，count 默认为所有
  - 可以直接在字符串上调用方法例如 print("hello".upper())
- **转义字符 format string**
  - **\n**:换行 newline
  - **\t**:tab，str.expandtabs(n)把 tabsize 改编为 n
  - ****:\
  - **\a**:铃声（控制台）
  - **\r**:回到当前行的开始（首字符）
    - print("This is a great world.\r Welcome here."): Welcome here.t world
  - **\b**:光标回退一位
  - **\f**:光标换到下一行的相同位置
  - print("**\"**"):"
  - 在字符串前加 **r 取消转义字符** r"…"
  - 如果写 print("\ ")，那么 python 会识别为\\并且不会报错，但会报一个警告(syntaxwarning)。
- **三种格式化输出方式**
  - "… %s… %d" % (a, b)和“{}. ..{}”.format(x, y)和: f“{x}…{y}”前后中间没有空格
  - **“…%d…%d…” %(age, distance) (注：来源于 C 语言)**
    - %s (字符串), %d(整数), %f(浮点数)
  - **“…{}…{}…”.format(age, distance) (注：python 所特有，更安全)**
    - {} 占位符，系统根据输入的数据自动推导其类型
      - **使用关键字参数的字符串格式化**
        - formatted_string = "这是一段包含占位符的字符串：{a}, 以及另一个占位符：{b}".format(a="苹果", b="橙子")
      - **使用位置参数的字符串格式化**
        - formatted_string = "这是第二个值：{1}, 这是第一个值：{0}".format("苹果", "橙子")
        - print(formatted_string)：这是第二个值：橙子，这是第一个值：苹果
      - **混合使用关键字参数和位置参数，位置参数要放在关键字参数前面**
      - **不使用参数**
        - “…{}…{}…”.format(age, distance)
    - format() 是 string 类自带的一个函数 ({}中间没有空格)
    - 要打出{}使用{{}}
    - 转换为二进制，box 分别为 2,8,16 进制

```
str1 = "0101"
x1 = int(str1)
x2 = int(str1, base=2)
print(x1, x2)

str1 = "{:b}".format(x1)
str2 = "{:b}".format(x2)
print(str1, str2)
```

```
101 5
1100101 101
```

```
str1 = "{:o}".format(x1)
str2 = "{:o}".format(x2)
print(str1, str2)

str1 = "{:x}".format(x1)
str2 = "{:x}".format(x2)
print(str1, str2)
```

```
145 5
65 5
```

-

```
txt1 = "My name is {fname}, I'am {age}".format(fname = "John", age = 36)
txt2 = "My name is {0}, I'am {1}".format("John",36)
txt3 = "My name is {}, I'am {}".format("John",36)
print(txt1)
print(txt2)
print(txt3)
```

```
My name is John, I'am 36
My name is John, I'am 36
My name is John, I'am 36
```

- **f-string: f"{age}...{distance}"**
  - 大写 F 也可以
  - print(f"The value of pi is approximately {math.pi:.3f}.")
  - print(f"{{{70+4}}}"):{74}
  - print(f"{{{{70+4}}}}"):{{70+4}}

string 格式

- {:format}：Alignment(对齐方式: <, =, >), width(宽度), 小数位
- 每个单元格，有**宽度。**所谓宽度(width)为 10，就是输出的时候，用当前光标位置后面 10 个位置来输出数据。有三种情况：
  - 数据长度大于 10。默认输出原数据
  - 小于 10。**左对齐<和右对齐>和居中对齐^**（默认右对齐）
  - 等于 10。 正常输出即可
- 对于浮点数，要进一步指明**小数点后面的位数。**10.3f 就是宽度为 10，后面保留 3 位, f 表示浮点数
- **f: 浮点数 d: 10 进制整数**
  - print("The lengths are {:>10.3f}, {:<10.3f}, {:^10.3f}.".format(10.1, 8.5, 6.9))输出 The lengths are 10.100, 8.500 , 6.900 .

```
print('{:>10}'.format('SJTU'))
print('{:<10}'.format('IEEE'))
print('{:10.3f}'.format(3.141592653589793))
```

```
      SJTU
IEEE
     3.142
```

  - {:.2f}
- **c: 字符**
  - print("The ASCII/unicode character is {:c}".format(65))输出 The ASCII/unicode character is A
- **s: 字符串**
  - print("The length is {:12s}".format("Hello"))输出 The length is Hello
- 常用列表

:<	Left aligns the result (within the available space)
:>	Right aligns the result (within the available space)
:^	Center aligns the result (within the available space)
:=	Places the sign to the left most position
:+	Use a plus sign to indicate if the result is positive or negative
:-	Use a minus sign for negative values only
:	Use a space to insert an extra space before positive numbers (and a minus sign before negative numbers)
:,	Use a comma as a thousand separator
:_	Use a underscore as a thousand separator
:b	Binary format
:c	Converts the value into the corresponding unicode character
:d	Decimal format
:e	Scientific format, with a lower case e
:E	Scientific format, with an upper case E
:f	Fix point number format
:F	Fix point number format, in uppercase format (show inf and nan as INF and NAN)
:g	General format
:G	General format (using a upper case E for scientific notations)
:o	Octal format
:x	Hex format, lower case
:X	Hex format, upper case
:n	Number format
:%	Percentage format

- **字典序**
  - >=<：字典序 ax>abx 逐个比较，如果到了一个字符最后一个位置，另一个字符还有，那短的字符小
- **加法乘法**
  - str1 +str2, str1*n, str1+=str2
- **三引号："""..."""**
  - 多行（也就是输入的时候有换行），会自动换行，加\取消换行
  - 在 Python 中，文档字符串（docstring）是一种字符串文字，它出现在模块、函数、类或方法定义中的第一个语句处。成为文档的特殊属性 __doc__。
- 控制台中输入东西自动有输出，而 IDE 必须要 print

Class

- **基本语法（class ...: ...）**
- **属性(a.x)**
- **成员函数(a.f())**
  - 第一个参数默认是 self，名字不一定，可以叫别的
  - 如果不是成员函数，也就是没有 self 参数，那么调用的时候会出错
    - class Test:def add(u, v):return u + v。当调用的时候如果使用 a.add(1,2),那么传入了三个参数报错，使用 Test.add(1,2)是可以的
- **构造函数（__init__()）**
  - 通过运行 self.a 增加新建对象 self 的属性 a
  - 不 return 或者值 return None
- **class variable 类变量**
  - 类变量（class variable）是属于类本身的变量，而不是属于某个特定实例的变量。类变量由所有类的实例共享，即所有实例访问相同的变量。下图 number 就是类变量

```python
class ClassVariable:
    number = 0

    def __init__(self):
        print("init")
        ClassVariable.number += 1 # don't use self
        self.x = 1

    @classmethod
    def print_num(cls):
        print(cls.number)

print(ClassVariable.number)
ClassVariable.print_num()

v1 = ClassVariable()
v2 = ClassVariable()
print(v1.number, v2.number)

print(ClassVariable.number)
ClassVariable.print_num()
```

```
0
0
init
init
2 2
2
2
```

- **@classmethod, cls**
  - @classmethod 是一种装饰器，用于将一个方法定义为类方法。与实例方法不同，类方法不绑定到实例上，而是绑定到类本身。类方法的第一个参数通常命名为 cls，表示调用该方法的类。

```python
class MyClass:
    class_variable = "类变量"

    def __init__(self, instance_variable):
        self.instance_variable = instance_variable

    @classmethod
    def class_method(cls):
        print("这是一个类方法")
        print("访问类变量:", cls.class_variable)
        cls.other_class_method()

    @classmethod
    def other_class_method(cls):
        print("这是另一个类方法")

# 调用类方法
MyClass.class_method()
```

- **__str__(), __repr__()**
  - **__str__()**用于 print
    - def __str__(self):　　return f"MyClass with value {self.value}"
  - **__repr__()**，用于开发者调试，def __repr__(self)　return...如果没有定义 str，会自动使用 repr
    - def __repr__(self):　　return f"MyClass({self.value})"
    - print(repr(instance))
- **__le__()__eq__(), 等比较函数**
  - object.__lt__(self, other)　# x<y
    - x<y 调用 x.__lt__(y)
  - object.__le__(self, other)　# x<=y
  - object.__eq__(self, other) # x==y
  - object.__ne__(self, other) # x!=y
  - object.__gt__(self, other)　# x>y
  - object.__ge__(self, other) # x>=y
- **__add__(), __radd__(), __iadd__() 等算术函数**
  - 对于 x

    - __add__: y = x + 1，x 在左侧调用
    - __radd__: z = -9 + x，x 在右侧调用
    - __iadd__: x += 100
    - __truediv__ ： 对应/
    - __floordiv__ ： 对应//
- **__getitem__()， __setitem__()**
    - 通过序号获取一个元素
    - __getitem__(self, index)
    - __setitem__(self, index, value)
- **一些内省工具**
    - print(instance.__class__) 给出 instance 的类
    - print(MyClass.__name__) 给出类的名称
    - print(SubClass.__bases__)给出超类的序列
    - print(instance.__dict__)给出对象所有属性的字典
    - print(dir(MyClass))返回对象所有属性和方法的列表
- **一些特殊函数**
    - **__copy__()** 用于 shallow copy
    - **__deepcopy__()** 用于 deep copy
    - **__hash__()** 哈希函数，实现了即 hashable
- **继承的机制（mro 不考）**
    - class DerivedClassName(BaseClassName1, BaseClassName2, ...)
    - 子类拥有超类所有的属性和方法
    - overriding 子类可以**重写**一个函数，覆盖超类的函数，而父类修改的话子类继承也会修改
    - 子类在继承的时候如果重写 init 的时候必须要先调用父类的 init 函数，否则就没有父类的 init 属性
    - **issubclass(a，b)**判断 a 是不是 b 的子类
    - **isinstance(a，b)**判断 a 是不是 b 的对象，a 如果是子类的对象，那也是超类的对象
    - 所有的类都继承了 **object 类**，所有的对象都可以调用 len(), print(), id(), type()
    - 一个类继承多个类，按顺序从左向右依次继承。多继承中，如果父类有同名属性或方法，先继承的优先级高于后继承的
- 一个传参的例子
    - class T:
    - def __init__(self, a, b):
    - self.a = a
    - self.b = b
    - a = T(1, 1)
    - b = a
    - b.a, a.b = 2, 3
    - print(a.a, b.b)输出 2 3

Module

- **module import 的语法（import ..., from ... import ..., 别名）**
  - **import module1**
    - module1.printer（‘Hello world!’）
  - **from moduel1 import printer，name2** 只调用了两个函数，别的函数没有调用
    - printer('Hello world!’）
  - **from module1 import *** 导入所有函数
    - printer('Hello world!’）
  - import modulename **as** name 原来的 modulename 会**被删除**
  - from modulename import attrname **as** name
- **导入顺序**
  - 标准库
  - 第三方库
  - 本地库
- **__name__与’ __main__**
  - 如果是 top level 文件，__name__ 被设置为 "__main__"
  - 如果是被 import 的文件，__name__被设置为模块名
- Module import 会把 import 的 module 都执行一遍，即使是 from xxx import xxx 也会
- 除非 reload，否则多次 import 只 import 一次。
- from importlib import reload

Exception

- **基本语法：try ... except ... else ... finally ...**
  - try:
  - # 可能会引发异常的代码
  - except SomeException as e:
  - # 处理异常的代码
  - else:
  - # 如果没有引发异常，执行这部分代码
  - finally:
  - # 无论是否引发异常，都会执行这部分代码
  -

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops! That was no valid number. Try again...")
```

```python
try:
    raise NameError("HiThere")
except NameError:
    print("An exception flew by!")
    raise  # try # raise?
```

- finally 一定会被执行（即使出现一些极端情况比如在循环里面并且前面有 break 或 continue）。
- 如果 try 没有引发 exception，那么 else 会被执行
- 可以用 raise 在 except 之后重新引发异常
- 可以用 except 处理多个异常 except (RuntimeError, TypeError, NameError):
- 最后一个 except 可以忽略 exception name，处理所有的 except
- **熟悉基本的异常类型**
  - **NameError**：尝试访问未定义的变量时引发。
  - **UnboundLocalError**：在函数中引用一个局部变量，但该变量在赋值前被引用。
  - **IndexError**：尝试访问列表、元组或字符串中不存在的索引。
  - **TypeError**：操作或函数应用于不支持的类型时引发。
  - **ValueError**：操作或函数接收到参数类型正确但值不合适时引发。
  - **KeyError**：尝试访问字典中不存在的键时引发。
  - **ZeroDivisionError**：尝试除以零时引发。
  - **AssertionError**：断言语句失败时引发。
  - **AttributeError**：尝试访问对象中不存在的属性或方法时引发。

Assertion

- **基本语法与意义**
  - assert condition, message　如果 condition 不是 True 会引发 AssertionError，message 可选，提示错误信息
  -

```python
def test_assertion(r):
    assert 0 < r < 1, f"{r=} is expected to be between 0 and 1."
    print(f"{r=} is valid.")


test_assertion(0.1)
test_assertion(1)  # AssertionError
```

0.0s

```
r=0.1 is valid.
AssertionError: r=1 is expected to be between 0 and 1.
```

- **禁用 assert**
  - 通过 -O or -OO 运行 python
  - Python 中，__debug__ 的默认值是 True，且一旦 Python 解释器运行后就无法更改这个值。__debug__ 的值取决于 Python 的运行模式：正常模式下值为 True，优化模式下值为 False。

assert 语句在正常模式下会启用，即当 __debug__ 为 True 时执行断言。断言失败时会引发 AssertionError 异常。

- 设置系统变量 set the system variable PYTHONOPTIMIZE to an appropriate value

- 执行 pytest test_samples.py

Random

- **random.random()**
  - 返回[0.0,1.0)之间的一个随机浮点数
- **random.randint(a,b)**
  - 返回 a 和 b 之间的一个随机整数，包括 ab
- **random.randrange(start, stop, step)**
  - 返回一个随机选择的整数从 range(start, stop, step)，不包括 stop
  - random.randrange(stop)生成 0 到 stop-1 之间的随机整数
  - Random.randint(a, b) = random.randrange(a, b+1)
- **random.uniform(a,b)**
  - 返回 a,b 之间的一个随机浮点数 N，a <= N <= b for a <= b and b <= N <= a for b < a,b 可能包含也可能不包含
- **random.seed()**
  - 初始化随机数生成器·，随机数生成器需要一个数字（seed value）来生成随机数，seed 默认使用当前系统时间。seed()可以定制 seed value，如果 seed 被设置为相同，那么获得的随机数相同（重复执行一个代码获得的值不变，写两遍代码执行一次获得的值是会不同的）
  - seed 必须是 None, int, float, str, bytes, bytearray。None 对应默认的 system time，除此以外如果不是整数，会转化为整数。
  - random.seed(10)直接使用
- **random.choice(list)**
  - 从一个非空序列中随机选择一个数 random.choice(lst)
- **random.choices(population, weights=None, *,cum_weights=None, k=1)**
  - 返回一个大小为 k 的列表
  - population：要从中选择元素的群体（例如列表、元组等）。
  - weights：一个可选的权重序列，用于指定选择每个元素的概率。如果未指定，所有元素被选择的概率相等。
  - cum_weights：一个可选的累计权重序列，与 weights 参数互斥。
  - k：要选择的元素数量。默认为 1。
  - *表示*之后只能用关键字参数不能用位置参数
  - 可能会重复，即使 population 中没有重复
- **random.shuffle(list)**
  - 随机打乱一个列表
- **random.sample(population,k)**

- 从给定的 population 序列或集合中随机选择 k 个唯一的元素，并返回一个包含这些元素的新列表。这些元素是唯一的，并且不会重复（如果 population 里面没有重复）。k 如果大于 population 中元素数量的话会引发 ValueError

File

unicode

- Unicode 表中的前 128 个字符与 **ASCII** 字符完全对应，A<a，ASCII 一共有 256 个（包括拓展），一个码不用所以可以看成 255 个

码值	控制字符	码值	控制字符	码值	控制字符	码值	控制字符	码值	控制字符	码值	控制字符	码值	控制字符	码值	控制字符
0	NUL	32	(space)	64	@	96	、	128	Ç	160	ó	192	└	224	α
1	SOH	33	!	65	A	97	a	129	ü	161	í	193	┴	225	ß
2	STX	34	"	66	B	98	b	130	é	162	ó	194	┬	226	Γ
3	ETX	35	#	67	C	99	c	131	â	163	ú	195	├	227	π
4	EOT	36	$	68	D	100	d	132	ä	164	ñ	196	─	228	Σ
5	ENQ	37	%	69	E	101	e	133	à	165	Ñ	197	┼	229	σ
6	ACK	38	&	70	F	102	f	134	å	166	ª	198	╞	230	µ
7	BEL	39	,	71	G	103	g	135	ç	167	º	199	╟	231	τ
8	BS	40	(	72	H	104	h	136	ê	168	¿	200	└	232	Φ
9	HT	41	)	73	I	105	i	137	ë	169	⌐	201	┌	233	Θ
10	LF	42	*	74	J	106	j	138	è	170	¬	202	┴	234	Ω
11	VT	43	+	75	K	107	k	139	ï	171	½	203	┬	235	δ
12	FF	44	,	76	L	108	l	140	î	172	¼	204	├	236	∞
13	CR	45	-	77	M	109	m	141	ì	173	¡	205	─	237	φ
14	SO	46	.	78	N	110	n	142	Ä	174	«	206	┼	238	ε
15	SI	47	/	79	O	111	o	143	Å	175	»	207	┴	239	∩
16	DLE	48	0	80	P	112	p	144	É	176	░	208	┴	240	≡
17	DCI	49	1	81	Q	113	q	145	æ	177	▒	209	┬	241	±
18	DC2	50	2	82	R	114	r	146	Æ	178	▓	210	┬	242	≥
19	DC3	51	3	83	S	115	s	147	ô	179	│	211	└	243	≤
20	DC4	52	4	84	T	116	t	148	ö	180	┤	212	Ô	244	⌠
21	NAK	53	5	85	U	117	u	149	ò	181	┤	213	╒	245	⌡
22	SYN	54	6	86	V	118	v	150	û	182	┤	214	┌	246	÷
23	TB	55	7	87	W	119	w	151	ù	183	┐	215	┼	247	≈
24	CAN	56	8	88	X	120	x	152	ÿ	184	┐	216	┼	248	≈
25	EM	57	9	89	Y	121	y	153	Ö	185	┤	217	┘	249	·
26	SUB	58	:	90	Z	122	z	154	Ü	186	│	218	┌	250	•
27	ESC	59	;	91	[	123	{	155	¢	187	┐	219	■	251	√
28	FS	60	<	92	\	124	|	156	£	188	┘	220	▄	252	ⁿ
29	GS	61	=	93	]	125	}	157	¥	189	┘	221	▌	253	²
30	RS	62	>	94	^	126	~	158	₧	190	┘	222	▐	254	■
31	US	63	?	95	_	127	DEL	159	ƒ	191	┐	223	▀	255	ÿ

- **Unicode** 规定了所有字符的序号，但这个序号不代表编码。譬如学生的序号和学号不是一样的。编码设计是一个复杂的问题，就要考虑时间效率，也要考虑空间占用率
- **UTF-8** 编码：字符代码小于 128 的字符表示为一个字节；代码在 128 和 0x7ff (2047) 之间的字符被转换为 2 个字节，每个字节的值在 128 到 255 之间；代码高于 0x7ff 的字符被转换为 3 或 4 个字节序列，值在 128 到 255 之间。
- **UTF-16 和 UTF-32**：分别以固定大小的 2 个字节或 4 个字节格式化文本，即使对于可以适合一个字节的字符也是如此。
- **"\xNN"**：十六进制字节值转义，表示单个字节
  - hex_escape = "\x48\x65\x6c\x6c\x6f" # 对应 "Hello"
  - print(hex_escape) # 输出: Hello
- **"\uNNNN"**：Unicode 转义，使用四个十六进制数字编码一个 2 字节（16 位）的字符码点。
  - unicode_escape = "\u4f60\u597d" # 对应 "你好"
  - print(unicode_escape) # 输出: 你好
- **"\UNNNNNNNN"**：Unicode 转义，使用八个十六进制数字编码一个 4 字节（32 位）的字符码点。
  - unicode_escape_long = "\U0001f600" # 对应
  - print(unicode_escape_long) # 输出:
- **字节字符串**（bytes 对象）仅支持十六进制转义序列"\xNN"用于编码文本和其他基于字节的数据
- 可以在第一行添加注释# -*- coding: UTF-8 -*-来制定编码方式，这样可以在 Python 脚本中方便地处理和显示各种非 ASCII 字符。

- 可以使用 Unicode 数据库中的字符名来表示特定字符
  - char_name = "\N{LATIN SMALL LETTER A}" # 对应字符 'a'
  - print(char_name) # 输出: a
- 一些函数
  - **ascii()**：返回对象的 ASCII 表示。
  - **bin()**：将整数转换为二进制字符串。
  - **oct()**：将整数转换为八进制字符串。
  - **hex()**：将整数转换为十六进制字符串。
  - **bytes()**：从各种源创建字节对象。
  - **chr()**：返回整数对应的字符。
  - **ord()**：返回字符对应的整数值。

byte sequence 语法以及意义

- **btxt = b'hello world'**
  - 字节对象是单个字节的不可变序列。字节字面量的语法与字符串字面量基本相同，只是前面添加了一个 b
- **类 bytes([source[, encoding[, errors]]])**
  - 字节字面量中只允许 ASCII 字符（不论声明的源代码编码为何）。任何超过 127 的二进制值必须使用适当的转义序列输入（\b、\o、\x）。
    - \x：表示十六进制转义，如 \x20 表示空格。
    - \b：表示八进制转义，如 \141 表示字符 a。
    - \o：表示八进制转义
  - 与字符串字面量一样，字节字面量也可以使用 r 前缀来禁用转义序列的处理。
  - source：可以是一个字符串、字节对象或一个可迭代对象。
  - encoding：如果 source 是字符串，必须指定编码。
  - errors：可选的错误处理方案，例如 'strict'、'ignore'、'replace'
- 由于两个十六进制数字正好对应一个字节，因此十六进制数是描述二进制数据的常用格式。
- **类方法 fromhex(string)**
  - 这个 bytes 类方法返回一个字节对象，解码给定的字符串对象。字符串每字节必须包含两个十六进制数字，ASCII 空白字符将被忽略。
  - hex_string = "48656c6c6f20576f726c64" byte_seq = bytes.fromhex(hex_string) print(byte_seq) 输出: b'Hello World'
- **hex([sep[, bytes_per_sep]])**
  - 返回一个字符串对象，其中包含实例中每个字节的两个十六进制数字。
  - byte_seq = b'Hello World' hex_string = byte_seq.hex() print(hex_string) 输出: 48656c6c6f20576f726c64
- **bytearray 对象是 bytes 对象的可变对应对象**
  - 类 bytearray([source[, encoding[, errors]]])
  - byte_array = bytearray("hello world", "utf-8") print(byte_array) # 输出: bytearray(b'hello world')

    - # 修改可变字节对象 byte_array[0] = ord('H')　　print(byte_array) 输出: bytearray(b'Hello world')
    - 在.encode()和.decode()中，默认的编码参数是"utf-8"。
  - **str 和 bytes** 具有几乎相同的功能。
    - text = "Hello, World!"　byte_seq = text.encode('utf-8')　print(byte_seq) # 输出: b'Hello, World!'
    - byte_seq = b'Hello, World!'　text = byte_seq.decode('utf-8')　print(text) # 输出: Hello, World!
  - 会尝试将可显示的字节（ASCII 范围内的）打印为相应字符，其它仍是数字形式
- **shell**
  - 绝对路径 C:\windows\....\a.txt
  - 相对路径.\a\b.txt
    - 假设你的当前工作目录是 C:\Projects，那么相对路径 .\a\b.txt 指向的文件实际路径是 C:\Projects\a\b.txt
  - 通过 shell 运行 python filename.py
    - filename.py 应该在当前工作目录
    - 或者使用绝对路径 python C:\xxxx\....\filename.py
    - Shell 中，路径名有空格用 ‘’ 包起来处理
  - python filename.py parameter1 parameter2 parameter3 ...
- **module sys**
  - 提供了对由解释器使用或维护的一些变量的访问，并与解释器强烈交互的功能。它始终可用：
  - **操控 Python 运行时环境的不同部分**
    - sys.version：显示当前 Python 解释器版本号的字符串。
    - sys.argv：返回传递给 Python 脚本的命令行参数列表。
      - sys.argv[0] 包含脚本文件名 xxx.py；sys.argv[1] 包含第一个参数，而 sys.argv[2] 包含第二个参数。
    - sys.path：这是一个环境变量，用于搜索所有 Python 模块的路径。
      - sys.path[0]：当前文件的目录。绝对路径：sys.path[0] + 文件名
    - sys.exit：使脚本退出并返回到 Python 控制台或命令提示符。通常用于在异常生成时安全地退出程序。
    - sys.maxsize：返回变量可以取的最大整数值。
    - sys.stdin，sys.stdout 和 sys.stderr：标准数据流
- **Module: sandard streams**
  - 系统的高级用户都知道标准流，即标准输入、标准输出和标准错误。它们通常被称为管道。常用的缩写是 stdin、stdout 和 stderr。
  - 标准输入 (stdin)：通常连接到键盘，用于从用户那里接收输入。
  - 标准输出 (stdout) 和 标准错误 (stderr)：通常连接到你工作的终端（或窗口），用于显示输出和错误信息。
  - 这些数据流可以通过 Python 的 sys 模块中的同名对象来访问，即 sys.stdin、sys.stdout 和 sys.stderr。

- 使用示例包括 input() 获取输入，print() 打印输出，以及 sys.stderr 写入错误信息。

- **Module os 操作系统**
  - os 模块提供了一种便携的方式来使用与操作系统相关的功能。以下是一些常用的方法和属性：
  - os.name：返回当前操作系统的名称。
  - 环境变量：
    - os.environ：获取环境变量的字典。
    - os.getenv()：获取指定环境变量的值。
    - os.putenv()：设置环境变量的值。
  - 目录操作：
    - os.chdir()：改变当前工作目录。
    - os.getcwd()：获取当前工作目录。
    - os.mkdir()：创建单级目录。
    - os.makedirs()：递归创建多级目录。
  - 文件和目录操作：
    - os.remove()：删除文件。
    - os.rmdir()：删除单级空目录。
    - os.rename(src, dst)：重命名文件或目录。
    - os.startfile()：使用关联应用程序打开文件。
    - os.walk()：生成目录树下的所有文件和目录。
  - 路径操作：
    - os.path.basename()：获取文件名。
    - os.path.dirname()：获取目录名。
    - os.path.exists()：检查路径是否存在。
    - os.path.isdir() 和 os.path.isfile()：判断路径是目录还是文件。
    - os.path.join()：连接多个路径组件。
    - os.path.split()：拆分路径成目录和文件名。
- **File 对象**
  - 当程序运行时，其数据存储在随机访问内存（RAM）中。RAM 快速且廉价，但当程序结束或计算机关闭时，RAM 中的数据将会丢失。为了使数据在下次打开计算机并启动程序时可用，必须将其写入非易失性存储介质（如硬盘、USB 驱动器或 CD-RW）。
  - 非易失性存储介质上的数据：数据存储在介质上的命名位置，称为文件。
  - 处理文件：处理文件类似于使用笔记本。要使用笔记本，必须先打开它；完成后，必须关闭它。
  - 在 Python 中，文件模式可以分为文本模式和二进制模式：
    - 文本模式
      - 文本模式 表示使用 str 对象。
      - 文本模式下，文件内容按照 Unicode 编码解释，可以是默认平台编码或指定的编码。通过传入编码名称给 open 函数，可以强制转换各种类型的 Unicode 文件。
      - 文本模式文件还会执行通用换行符转换：默认情况下，所有换行形式都映射为脚本中的单个 \n 字符，无论运行平台如何。

        - 文本文件还处理某些 Unicode 编码方案中存储在文件开头的字节顺序标记（BOM）的读取和写入。
    - 二进制模式
        - 二进制模式 表示使用 bytes 对象。
        - 二进制模式文件返回的是原始文件内容，以表示字节值的整数序列形式返回，没有编码或解码，也没有换行符转换。
- 路径
    - Mac and Linux: C:/usr/xxxxx
    - Windows: C:\windows\xxxx\
    - os.path
        - os.path.join()：使用当前操作系统的正确斜杠来构建路径字符串。
        - 优点：兼容性好，能够根据不同操作系统自动选择适当的路径分隔符。
    - pathlib（推荐使用，Python 3.4 引入）
        - Pathlib 优点：
            - 使用正斜杠 / 即可，Path 对象会将其转换为当前操作系统的正确斜杠。
            - 可以直接在代码中使用 / 运算符来添加路径部分，无需反复输入 os.path.join(a, b)。
            - 提供面向对象的文件系统路径操作方法，更加直观和易用。

- **open()以及读取模式**
    - mf= open(file, mode='r', buffering=- 1, encoding=None, errors=None, newline=None, closefd=True,opener=None)
    - mode 默认是 read（r），file 文件名必须加“”
    - **r**: read
    - **w**: write 如果不存在会创造，如果存在会先清空原文件，再从头写入
    - **x**: exclusive creation 创建一个文件，如果已经存在会引发异常
    - **b**: binary
    - **+**: read and write
    - **a**: appending 在最后增加，不覆盖原内容
    - 模式可以混合
    - 常用模式

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

- **.close()以及 with 语法**
  - mf.close()
  - file.closed: 如果文件关闭返回 True，否则返回 False
  - file.mode: 返回打开文件的模式
  - file.name: 返回文件名称
  - 可以多行 with open('mylog.txt') as infile, open('a.out', 'w') as outfile:
- **.read(), .readline(), .readlines()**
  - **mf.read()**读取换行符，从上次读取位置继续读取
  - **mf.readline()** 方法返回一行“字符串”中的所有内容，直到并包括换行符 \n。换行符 \n 也会被读取。只读一行，需要循环
  - **mf.readlines()**：读取并返回文件中的行的列表。

```
rf = open("test.txt")
res = rf.readlines()
rf.close()
print(res)
```

```
['My second file written from Python\n', '--------------------------------\n', 'Hello, world, again\n', '\n', 'My third file written from Python\n',
```

  - **input()**：当输入 \n 时终止，并且 \n 会被忽略。
  - **mf.read(n) 方法**：从文件中读取 n 字节。调用 f.read(size) 读取文件内容，返回字符串（文本模式下）或字节对象（二进制模式下）。多次调用会连续读取

```
rf = open("test.txt", "rb")
while True:
    line = rf.read(10)
    print(line)
    if len(line) == 0:
        break
print()
rf.close()
```

```
b'My second '
b'file writt'
b'en from Py'
b'thon\r\n----'
b'----------'
b'----------'
b'---------\r'
b'\nHello, wo'
b'rld, again'
```

- 可以用 for 来 read 每一行
  - with open("test.txt", "r") as reader: for line in reader: print(line, end="")

- **.write(), .writelines()**
  - **mf.write(string)**把 string 的内容写入 file，返回写入字符的个数，写入的必须是 string
    - 不会添加换行符
    - items = ['abc', '123', '!@#'] items = map(lambda x: x + '\n', items)
      w.writelines(items)
  - **f.writelines (lines)**把一个行的 list 写入文件，必须是 str，不会加换行符，如果要换行在列表每个元素末尾加\n
- **.seek(), .flush()**
  - **f.tell()**返回整数告诉当前文件位置
  - **f.seek(offset, whence)**：用于改变文件对象的位置。位置通过将偏移量 offset 添加到参考点来计算；参考点由 whence 参数选择。whence 的值可以是：
    - 0：从文件开头开始测量。默认值
    - 1：使用当前文件位置。
    - 2：使用文件末尾作为参考点。
  - **f.flush()**：刷新文件对象的写缓冲区。将缓冲区中的数据立即写入硬盘
  -

```python
f = open('workfile', 'wb+')
f.write(b'0123456789abcdef')
f.seek(5)      # Go to the 6th byte in the file
print(f.read(1))
f.seek(-3, 2)  # Go to the 3rd byte before the end
print(f.read(1))
f.close()
```

```
b'5'
b'd'
```

- 混合读写
  - 先写后读要 flush()
  - 先读后写要 seek()

Iterable

iterator 的概念

- 迭代器是一个可以迭代的对象，意味着你可以遍历所有的值。

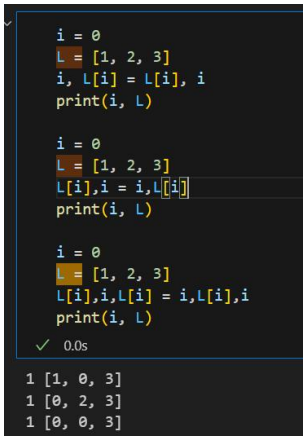
iter()与 next()

- 一个类，如果实现了__iter__(), __next()__函数，则可以通过 iter()来获得它的迭代器，从而配合 next()函数遍历它，实现连续访问
- 和这个类对应的 iterable (iterator): 迭代器、广义下标. 可以用 for 来遍历
- ： for x in collection_a: 给定后，修改 x 不改变 collection_a 和迭代顺序
- iter()+next() 获取 collection_a 的迭代顺序

```python
ist = iter(set((-1, -2, -3, -4)))
print(next(ist))
print(next(ist))
print(next(ist))
print(next(ist))

ir = iter(range(-4, 5, 2))
print(next(ir))
print(next(ir))
print(next(ir))
print(next(ir))
print(next(ir))

M = [
    [1, 2, 3],  # A 3 × 3 matrix, as nested lists
    [4, 5, 6],  # Code can span lines if bracketed
    [7, 8, 9],
]

g = iter([sum(row) for row in M])
print(next(g))
print(next(g))
print(next(g))
```

- 在 Python 中，某个对象是否是可迭代的（iterable），取决于它是否是以下两种之一：
  - **物理存储在内存中的序列**：比如列表、元组等。
  - **在迭代操作时逐个生成项目的对象**：这类对象不将所有项目一次性存储在内存中，而是在需要时动态生成项目，类似于“虚拟”序列。
  - 这两类对象被认为是可迭代的，因为它们支持迭代协议（iteration protocol）：
- **iter()** 调用会返回一个对象，该对象在接收到 **next()** 调用时会前进，并在完成生成值时引发一个异常。
- 生成器表达式其值不会一次性存储在内存中，而是在需要时生成，通常通过迭代工具来请求。
- 当使用迭代工具时，Python 文件对象会逐行迭代：文件内容不是以列表形式存储，而是按需获取。这两者在 Python 中都是可迭代对象。

- 迭代协议的重要性：每一个从左到右扫描对象的 Python 工具都使用迭代协议。这就是为什么在上一个部分中直接对字典使用 sorted 调用是有效的——我们不需要调用 keys 方法来获取序列，因为字典本身就是可迭代对象，并且 next 会返回连续的键。
  - my_dict = {'b': 2, 'a': 1, 'c': 3}
  - sorted_keys = sorted(my_dict)
  - print(sorted_keys) # 输出: ['a', 'b', 'c']
- 可迭代对象（Iterable）：是指一个支持 iter() 调用的对象。也就是说，可以对它调用 iter() 方法以返回一个迭代器。常见的例子包括列表、元组、字符串、字典等。
  - **迭代器（Iterator）**：是指一个支持 next() 调用的对象。它是由可迭代对象调用 iter() 方法返回的对象，并且支持 next() 方法，能够逐个返回值，直到没有值时引发 StopIteration 异常。
  - Range is iterable 但不是 iterator
  - 迭代器的迭代器是自己
  - for 循环会调用可迭代对象的 __iter__() 方法，以获取迭代器。然后在循环的每次迭代中，for 循环会调用迭代器的 __next__() 方法以获取下一个元素。当 __next__() 方法引发 StopIteration 异常时，for 循环会捕获该异常并只捕获这种异常，知道迭代已经完成，随后退出循环。
  - 系统自定义的 list, tuple 等实现了__iter__(),__next__()。自定义的类型，必须自己实现。
  - For 遍历过程中，自动调用__iter__(), __next__()。
  - 对于定义了__iter__(), __next__()的自定义类，可以通过 iter()生成迭代器， next()进行迭代，iter(), next()分别对应于__iter__(), __next__()
  - next()在读取 file 的时候移动一行而不是一个字符
  - 两个类型可以互相转换的一个必要条件是它们都是 iterable，a, b, c = x 前提是 x 能够 iterable



- iterator 的实现：__iter__()__iter__(), __next__()__next__() 以及 StopIteration 异常
  - __iter__()：该方法用于初始化迭代器，可以执行一些初始化操作，但必须返回迭代器对象本身。
  - _next__()：该方法用于返回序列中的下一个项目。每次调用 __next__() 方法时，应返回下一个项目。如果没有更多项目可返回，应引发 StopIteration 异常，防止迭代无限进行。
  - For 的过程，就是背后调用__iter__, __next__的过程
  - 迭代器不能直接 print 出来，会输出类似<list_iterator object at 0x7f8e4c5d1c70>，而需要用比如 list（）转化
  -

```python
class MyNumbers:
    def __init__(self, a=0):
        self.a = a

    def __iter__(self):
        self.a = 1
        print("__iter__")
        return self

    def __next__(self):
        if self.a <= 7:
            print("next")
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration


myclass = MyNumbers()

for x in myclass:
    print(x)
```

Iterator	Arguments	Results
product()	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
permutations()	p[, r]	r-length tuples, all possible orderings, no repeated elements
combinations()	p, r	r-length tuples, in sorted order, no repeated elements
combinations_with_replacement()	p, r	r-length tuples, in sorted order, with repeated elements

- This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple.

namedtuple()	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values
UserDict	wrapper around dictionary objects for easier dict subclassing
UserList	wrapper around list objects for easier list subclassing
UserString	wrapper around string objects for easier string subclassing

- **unpacking: *a, **b**
  - ***出现在赋值语句里**
    - *解包对象必须在一个 list 或者 tuple 里面
      - list1 = [1, 2, 3] list2 = [4, 5, 6]
      - combined = [*list1, *list2]
      - print(combined) # 输出: [1, 2, 3, 4, 5, 6]
    - *f 的 f 经过赋值以后是 list 类型
      - def func(x, y, z):
      - print(x, y, z)
      - lst = [1, 2, 3]
      - func(*lst) # 输出: 1 2 3
  - ***出现在函数调用里面**
    - *args 里面 args 是 tuple 类型
  - ****用于解包字典对象**
    - 例 1

        - dict1 = {'a': 1, 'b': 2}dict2 = {'c': 3, 'd': 4}combined = {**dict1, **dict2}print(combined) # 输出: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
    - 例 2
        - def func(a, b, c):
        -     print(a, b, c)
        - dct = {'a': 1, 'b': 2, 'c': 3}
        - func(**dct) # 输出: 1 2 3
    - 例 3
        - def greet(name, age):
        -     return f"Hello, my name is {name} and I am {age} years old."
        - person = {'name': 'Alice', 'age': 30}
        - message = greet(**person) # 相当于 greet(name='Alice', age=30)
        - print(message) # 输出: Hello, my name is Alice and I am 30 years old.
    -

```
# *f = lst  # SyntaxError: starred assignment target must be in a list or tuple
(*f, a) = tuple(lst)
(*f,) = tuple(lst)
```

- **all(iterable)**如果其中所有元素都是 true 返回 True
- **any(iterable)**如果其中有一个元素使 True 返回 True

Match

- 从上往下对比，如果遇到符合的 pattern，就执行该 case 的代码，结束后离开 match
- _ :表示 wildcard, 会 match 任意的情况，可以省略
-

```
def test_day(x):
    match x:
        case 1:
            print("Monday")
        case 2:
            print("Tuesday")
        case 3:
            print("Wednesday")
        case 4:
            print("Thursday")
        case 5:
            print("Friday")
        case 6:
            print("Saturday")
        case _:  # try to remove it.
            print("Sunday")


for x in range(1, 10):
    test_day(x)
```

- match command.split(): case [action, obj]:
    - **结构化匹配**，比如 match 对象与第一个 case[action, obj]结构匹配，那么就会 match 并且对应赋值，不匹配的话就执行其它 match
        -

```
match command.split():
  case ["quit"]:
    print("Goodbye!")
    quit_game()
  case ["look"]:
    current_room.describe()
  case ["get", obj]:
    character.get(obj, current_room)
  case ["go", direction]:
    current_room = current_room.neighbor(direction)
```

- **case ["drop", *objects]** 模式匹配任何以 "drop" 作为第一个元素的序列。剩余的所有元素会被捕获到列表 objects 中

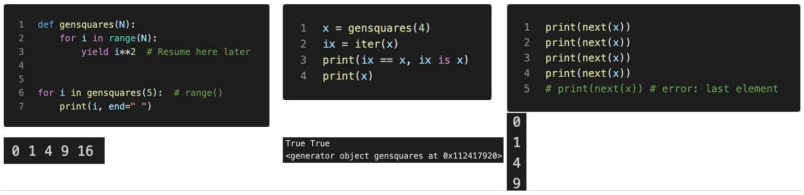
```
match command.split():
  case ["drop", *objects]:
    for obj in objects:
      character.drop(obj, current_room)
  # The rest of your commands go here
```

- 可以使用更高级的序列模式匹配 **case ["first", (left, right), _, *rest]**
  - 第一个元素 first，第二个元素元组，第三个元素忽略用占位符表示，rest 捕获其余元素
- **组合模式 case ["north"] | ["go", "north"]:**
  - 所有备选模式必须绑定相同的变量。例如，[1, x] | [2, y] 不被允许，因为这会导致成功匹配后绑定的变量不明确；而 [1, x] | [2, x] 是允许的，因为无论哪种情况，都始终会绑定 x
  - command = input("What are you doing next? ")
  - match command.split():
  - case ["go", ("north" | "south" | "east" | "west") as direction]:。。。
- **or 模式和 as 模式：** case ["go", ("north" | "south" | "east" | "west") as direction] 使用 or 模式验证命令中的方向，同时使用 as 模式将方向绑定到 direction 变量。
- **增加守卫条件**
  - case ["go", direction] if direction in current_room.exits 中，首先检查模式是否匹配。如果模式匹配，守卫条件才会进行检查，验证 direction 是否在 current_room.exits 中

Generator

- 基本语法
  - yield
    - 当一个函数包含 yield 语句时，它会成为一个生成器函数。生成器函数在执行时不会一次性返回一个值，而是会在每次遇到 yield 语句时暂停执行，并返回一个值。此时，函数的状态（包括局部变量的值和执行位置）会被保存。当生成器函数被恢复执行时，它会从暂停的地方继续执行，并保留之前的状态。Generator 是自己的 iterator

```
def gensquares(N):
    for i in range(N):
        yield i ** 2 # Resume here later
```



- yield 语句可以有多个

```
def gensquares(N):
  for i in range(N):
    yield i**2
  for i in range(N):
    yield i**3  # You can have multiple yield statements
```

- 生成器表达式的语法和列表推导式非常相似，支持所有的语法，包括 if 过滤器和循环嵌套。但它们是用圆括号 () 而不是方括号 [] 包围的。不加括号是语法错误
  - # 列表推导式
  - even_squares = [x * x for x in range(10) if x % 2 == 0]
  - print(even_squares) # 输出: [0, 4, 16, 36, 64]
  - # 生成器表达式
  - even_squares_gen = (x * x for x in range(10) if x % 2 == 0)
  - print(list(even_squares_gen)) # 输出: [0, 4, 16, 36, 64]
  - total = sum((x * x for x in range(10)))中一层圆括号可以省略
  -

```
lst = [x ** 2 for x in range(4)]
print(type(lst))

ge = (x ** 2 for x in range(4))
print(type(ge))

lst1 = list(x ** 2 for x in range(4))
print(type(lst1))

ge1 = x ** 2 for x in range(4) #error
print(type(ge))
```

```
sg = (x for x in "hello world. 苟利")
print(next(sg))
lis = list(sg)
print(lis)
#print(next(sg)) #error生成器在list()中已经被消耗，生成完了
```

- comprehension 语法
  - 元组推导，字典推导，集合推导，tuple 没有 comprehesion

```python
g = (2 * x + 1 for x in range(7))
lg = tuple(g)
print(type(g), type(lg), lg)

g = (2 * x + 1 for x in range(7))
lg = set(g)
print(type(g), type(lg), lg)

lg = {2 * x + 1: x for x in range(7)}
print(type(g), type(lg), lg)
```

✓ 0.0s

```
<class 'generator'> <class 'tuple'> (1, 3, 5, 7, 9, 11, 13)
<class 'generator'> <class 'set'> {1, 3, 5, 7, 9, 11, 13}
<class 'generator'> <class 'dict'> {1: 0, 3: 1, 5: 2, 7: 3, 9: 4, 11: 5, 13: 6}
```

- **函数高级**
  - **函数是 callable 的**
    - 可以通过 greeting.__call__()调用函数
    - print（callable（greeting））：True，而其它数据类型可能返回 False
    - 一个类要实现 call 方法才是 callable 的
  - **默认参数**
    - 函数的默认参数保存在 funcname.__defaults__中
    - 如果默认参数的类型可以被修改，那么默认参数可以变，而且所有的默认参数会一起变，你下次调用的时候默认参数就不是原来的了，故设置为 None 或者一个不会出现的值， None is immutable，只要是 immutable 的，就每次都会 reset

```python
def test_default(x, lst=[]):
    print(lst, type(lst), id(lst))
    lst.append(x)
    return lst


print(test_default.__defaults__[0])
print(test_default(1))
print(test_default(2))
print(test_default(3))
print(test_default(4))
print(test_default(5))
print(test_default.__defaults__[0])
```

```
[]
[] <class 'list'> 140425493656640
[1]
[1] <class 'list'> 140425493656640
[1, 2]
[1, 2] <class 'list'> 140425493656640
[1, 2, 3]
[1, 2, 3] <class 'list'> 140425493656640
[1, 2, 3, 4]
[1, 2, 3, 4] <class 'list'> 140425493656640
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

```python
def test_default(x, lst=None):
    print(lst, type(lst), id(lst))
    if not lst:
        lst = []

    lst.append(x)
    return lst


print(test_default.__defaults__[0])
print(test_default(1))
print(test_default(2))
print(test_default(3))
print(test_default(4))
print(test_default(5))
print(test_default.__defaults__[0])
```

```
None
None <class 'NoneType'> 4332715632
[1]
None <class 'NoneType'> 4332715632
[2]
None <class 'NoneType'> 4332715632
[3]
None <class 'NoneType'> 4332715632
[4]
None <class 'NoneType'> 4332715632
[5]
None
```

```python
def test_default(x, const=-1):
    print(f"Before {x = }, {const = }")
    const += x
    print(f"After {x = }, { const = }")


test_default(x=0)
test_default(x=1)
test_default(x=2)
test_default(x=3)
test_default(x=-1)
```

```
Before x = 0, const = -1
After x = 0,  const = -1
Before x = 1, const = -1
After x = 1,  const = 0
Before x = 2, const = -1
After x = 2,  const = 1
Before x = 3, const = -1
After x = 3,  const = 2
Before x = -1, const = -1
After x = -1,  const = -2
```

const每次都会reset

    - 如果不可以被修改，那么不变
    - 无论如何，id 不变
  - **__doc__=, __name__=**
    - __name__: 函数的名称。
    - __code__: 包含编译后的字节码和其他代码对象信息。
    - __doc__: 函数的文档字符串，通常用于描述函数的功能。
    - __dict__: 包含函数的属性和方法的字典。
    - 使用：print(func.__name__)
  - **可以自行增加函数对象的属性**
    - 在定义函数后增加属性
      -

```python
def my_function():
    return "Hello, world!"

# 向函数对象添加属性
my_function.description = "这是一个简单的函数"
my_function.version = 1.0

# 访问函数对象的属性
print(my_function.description)  # 输出：这是一个简单的函数
print(my_function.version)      # 输出：1.0
```

- property() 和@property 装饰器用于创建属性
-

Python Copy

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius

    def get_radius(self):
        return self._radius

    def set_radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    radius = property(get_radius, set_radius)

circle = Circle(5)
print(circle.radius)  # 输出：5
circle.radius = 10
print(circle.radius)  # 输出：10
# circle.radius = -1  # 引发 ValueError: Radius cannot be negative
```

Python Copy

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

circle = Circle(5)
print(circle.radius)  # 输出：5
circle.radius = 10
print(circle.radius)  # 输出：10
# circle.radius = -1  # 引发 ValueError: Radius cannot be negative
```

- **变元作用域：global, nonlocal, del**
  - nonlocal 在嵌套函数中使用，允许内层函数中引用外层函数的变量，而不是全局变量

```python
def outer_function():
    outer_var = "I am outer"

    def inner_function():
        nonlocal outer_var
        outer_var = "I am changed by inner"
        print(outer_var)

    inner_function()
    print(outer_var)

outer_function()
```

- Global, nonlocal, local 对同一个变量只存在一个状态

- **函数的嵌套定义**
  - 函数嵌套：内部函数、外部函数
  - 内部函数引用外部函数的数据
  - 内部函数被外部函数返回

```python
# # return value
def make(N):
    def action(x):
        return x ** N

    return action

f1 = make(2)
f2 = make(3)
f3 = make(4)

print(f1(3), f2(4), f3(5))
```

```
9 64 625
```

- **decorator 机制**
  - 函数闭包：被装饰函数作为参数传入
  - 内嵌函数调用被装饰函数，同时增加新的行为
  - 外部函数返回内嵌函数
  - 不改变被装饰函数代码的情况下，增广其功能
  - 也可以装饰第三方代码，还可以有*args, **kwargs，以及有返回值的代码

```python
def hello_decorator(func):

    def inner1():
        print("Hello, this is before function execution")
        func()
        print("This is after function execution")


    return inner1


@hello_decorator
def function_to_be_used():
    print("This is inside the function !!")


function_to_be_used()

@hello_decorator
def function_to_be_used_2():
    print("耗子尾汁")


function_to_be_used_2()
```

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
Hello, this is before function execution
耗子尾汁
This is after function execution
```

```python
# defining a decorator
def hello_decorator(func):

    # inner1 is a Wrapper function in
    # which the argument is called

    # inner function can access the outer local
    # functions like in this case "func"
    def inner1():
        print("Hello, this is before function execution")

        # calling the actual function now
        # inside the wrapper function.
        func()

        print("This is after function execution")

    return inner1


# defining a function, to be called inside wrapper
def function_to_be_used():
    print("This is inside the function !!")


# passing 'function_to_be_used' inside the
# decorator to control its behavior
function_to_be_used = hello_decorator(function_to_be_used)


# calling the function
function_to_be_used()
```

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

```python
from math import sin, cos

def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

sin = our_decorator(sin)
cos = our_decorator(cos)

for f in [sin, cos]:
    f(3.1415)
```

```
Before calling sin
9.265358966049024e-05
After calling sin
Before calling cos
-0.9999999957076562
After calling cos
```

```python
def hello_decorator(func):
    def inner1(*args, **kwargs):

        print("before Execution")

        # getting the returned value
        returned_value = func(*args, **kwargs)
        print("after Execution")

        # returning the value to the original frame
        return returned_value

    return inner1


# adding decorator to the function
@hello_decorator
def sum_two_numbers(a, b):
    print("Inside the function")
    return a + b

a, b = 1, 2

# getting the value through return of the function
print("Sum =", sum_two_numbers(a, b))
```

```
before Execution
Inside the function
after Execution
Sum = 3
```