CSCI 2040 Python

①②③④⑤⑥⑦⑧⑨⑩

slash / backslash 斜杠 / 反斜杠

> `/`：斜杠，分隔作用
>
> `\`：反斜杠，表示目录

placeholder 占位符

explicit positional index 显式位置索引

delimit 限定，划定界限

subscripted notation 下标记号

Immutable 不可变的

retrieve 检索

Iterable 可迭代的

Iterable object 可迭代对象

Iterator 迭代器

enumerate 枚举

implicitly 隐式

serialization 序列化（`Pickle` 概念）

dump 转储（序列化函数）

programming paradigms 编程范式

Inheritance 继承

Introduction

input & print

> age = int(input("xxxx"))，强制类型转换。
>
> print 会把 int 参数先转换为 str 再输出。

Python 的变量不用声明数据类型，它会自己检测.

`print` 自带换行符.

多个参数输出在同一行有两种方式：逗号（中间有空格隔开）或加号（串联，中间无字符）

```
message = "Hello to the wonderful world of Python"
message2 = "and welcome to CSCI2040"
print(message)
print(message2)
print(message, message2)
print(message + message2)
```

控制台输出：

```
Hello to the wonderful world of Python
and welcome to CSCI2040
Hello to the wonderful world of Python and welcome to CSCI2040
Hello to the wonderful world of Pythonand welcome to CSCI2040
```

Comments 注释

> python 的注释用 `#`

Lecture 1: Basics

1.1 变量命名

> 字母、数字、下划线组成。不能以数字开头.
>
> 注意，大写 "I", "O" 和数字 "1", "0" 可能混淆.
>
> 变量中不能有空格，使用下划线隔开单词.

Reserved words / keywords 保留字

> 不能用于变量命名。

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	break
except	in	raise		

1.2 Strings 字符串

Change the case

```
first_name = 'go'
print(first_name) # just print the string
print(first_name.title()) # capitalize the first letter
print(first_name.upper()) # capitalize the whole string

last_name = "Lamp"
print(last_name.lower()) # Lower case as well
```

控制台输出：

```
go
Go
GO
lamp
```

字符串可以直接用加号串联 (concatenation)

```
full_name = first_name + last_name
```

> 串联是直接连在一起，中间没有逗号隔开.
>
> 注意串联只能串联两个字符串，不能和其他类型（例如int）串联.
>
> 但是 `print()` 中逗号两侧可以是不同类型，输出会用空格隔开

```
first_name = 'go'
print ("Our king's name is = " + first_name , 1)
```

控制台输出：

```
Our king's name is = go 1
```

逗号和加号混用可读性较差，但是又想用空格隔开，可以这么处理：

```
first_name = 'go'
last_name = "Lamp"
full_name1 = first_name.title() + " " + last_name   # 注意这里的 + " " +
print ("Our king's full name is = " + full_name1)
```

控制台输出：

```
Our king's full name is = Go Lamp
```

control characters 控制字符

制表符 `\t`

```
first_name = 'go'
last_name = "Lamp"
full_name1 = first_name.title() + " " + last_name
print(full_name1, "is a jerk") # automatically add one space
print(full_name1, "\tis a jerk") # adding a tab
print(full_name1, "\t\t\tis a jerk!!!!") # adding 3 tabs
```

> \t 加 tab , 2/4/8 个英文字符单位（不同的 `IDE` 可能有不同的制表符宽度）.

控制台输出：

```
Go Lamp is a jerk
Go Lamp     is a jerk
Go Lamp             is a jerk!!!!
```

> 这里的制表符在 IDE 里占 8 个单位，但是移植到这个 .md 里就变成了 4 个单位

New line 换行符 `\n`

```
first_name = 'go'
last_name = "Lamp"
full_name1 = first_name.title() + " " + last_name
print(full_name1, "\nis a jerk")
print("\n",full_name1, "\n\t\tis a jerk")
```

控制台输出：

```
Go Lamp
is a jerk

 Go Lamp
        is a jerk
```

> 中间的空行是第一个 `print()` 自带的换行和第二个 `print()` 的换行符叠加形成

Stripping white spaces 去空格

> strip：
>
> noun. 条，带
>
> verb. 剥夺，脱掉，扒，卸，劈

```
name = " CUHK " # a string with white spaces, before and after
print("stripping left of the string, name=" + " ***" + name.lstrip() + "***")
print("stripping right of the string, name=" + " ***" + name.rstrip() + "***")
print("stripping both sides of the string, name=" + " ***" + name.strip() +
"***")
```

控制台输出：

```
stripping left of the string, name= ***CUHK ***
stripping right of the string, name= *** CUHK***
stripping both sides of the string, name= ***CUHK***
```

`type()` and `dir()`

> `type()`: 返回类型
>
> `dir()`: 返回一个列表，包含字符串对象的所有属性和方法。

注意，如果希望在 .py 脚本中看到这些输出，需要使用 print() 函数

```
stuff = 'hello world'
print(type(stuff))
print(dir(stuff)) # show all built-in functions of tyis type
help(stuff.capitalize) # show the meaning of the capitalize function
```

控制台输出：

```
<class 'str'>
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
Help on built-in function capitalize:

capitalize() method of builtins.str instance
    Return a capitalized version of the string.

    More specifically, make the first character have upper case and the rest
lower
    case.
```

> 注意 `help()` 在内部实现了控制台输出，函数本身的返回值是 `None`，不需要 `print`.

1.3 数字类型

> Numbers and numerics
>
> 注意，数字类型 ≠ 数据类型.

`int`

> Integers，十进制整数

```
print (1+2.) # addition
print (3.-2) # subtraction
print (5*2.) # multiplication
print (8/4) # division 注意单斜杠是浮点数除法，双斜杠才是整数除法. 另外注意这里的斜杠是正斜杠
print (2.0**4) # exponentiation
print (2+3*4)
print ((2+3)*4)
```

```
3.0
1.0
10.0
2.0
16.0
14
20
```

`float`

> Floating points，浮点数

```
print (0.1+0.1) # 输出: 0.2
print (0.1+0.2) # 输出: 0.30000000000000004
```

```
0.2
0.30000000000000004
```

> In Python 2.7, print 3/2 will give you 1
> In Python 3.X, print 3/2 will give you 1.5
> There are more differentiation between 3.X and 2.X for Python

operators: `+`, `-`, `*`, `/`, `//`, `%`

```
print(3+5) # adding two integers (输出: 8)
print(3. + 5) # type conversion (输出: 8.0)
print('Aaa' + 'Bbbbbb' +'CCC') # concatenation (串联)
print('Aaa' , 'Bbbbbb' ,'CCC') # 输出: Aaa Bbbbbb CCC
```

```
8
8.0
AaaBbbbbbCCC
```

> 注意「加号串联」和「逗号隔开」不同，加号串联不会在中间添加空格.
>
> `//`：整数除法，返回整数部分（向下取整）
>
> `%`：取模运算符，用于计算两个数字相除后的余数.

```
print(-5.2) # gives a negative number
print(5-3)
print(2*3)
print('Lu'*2) # generate character strings
print(3**4) # power
print (3.0**4.)
print(5/3)
print(5/3.)
print(4//3) # gives floor
print(5.0//3.)
print (17 % 3) # gives remainder
print (17. % 3)
```

```
-5.2
2
6
LuLu
81
81.0
1.6666666666666667 # 注意到单斜杠就是浮点运算，即使两个操作数都是整数
1.6666666666666667
1  # python 的双斜杠 // 和 C 的单斜杠一致
1.0
2
2.0
```

Some funny cases:

```
print(4/2)  # 输出 2.0
```

operators: `<`, `>`, `<=`, `>=`, `==`, `!=`, `not`, `and`, `or`

Operator	Description	Examples
<	Less than	5 < 3 gives 0 (i.e. False) and 3 < 5 gives 1 (i.e. True). Comparisons can be chained arbitrarily: 3 < 5 < 7 gives True.
>	Greater than	5 > 3 returns True. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns False.
<=	Less than or equal to	x = 3; y = 6; x <= y returns True.
>=	Greater than or equal to	x = 4; y = 3; x >= 3 returns True.

Operator	Description	Examples
==	Equal to	x = 2; y = 2; x == y returns True. x = 'str'; y = 'stR'; x == y returns False. x = 'str'; y = 'str'; x == y returns True.
!=	Not equal to	x = 2; y = 3; x != y returns True.
not	Boolean NOT	x = True; not y returns False.
and	Boolean AND	x = False; y = True; x and y returns False since x is False. In this case, Python will not evaluate y since it knows that the value of the expression will has to be false (since x is False). This is called short-circuit evaluation.
or	Boolean OR	x = True; y = False; x or y returns True. Short-circuit evaluation applies here as well.

*数字类型 vs 数据类型

> 不考

注意，数字类型 ≠ 数据类型. Python 的数据类型包括数字类型（`int`，`float`，`complex`），序列类型（`str`，`list`，`tuple`，`range`），映射类型（`dict`），集合类型（`set`，`frozenset`），布尔类型（`bool`），None 类型（`None`），二进制类型（`bytes`，`bytearray`，`memoryview`）.

Python 数据类型的本质都是类，对应的变量/常量都是类的对象（实例）. 上面列出的是它内置的类，开发者也可根据需要自己编写/继承/拓展.

这点和 C 有很大不同，C 是面向过程语言，数据类型表示内存中实际数据的类型，即数据存储的形式.

C++ 兼容 C 语言，同时支持面向对象，因此 C++ 的数据类型分为两类：基本数据类型（来自 C）和用户定义的类型（类）.

Python，C，C++ 的区别：Python 所有数据类型都是类，C 所有数据类型都不是类，C++ 部分是类部分不是类.

这么看编程语言的发展趋势很有逻辑：由硬件发展和市场需求推动，一开始计算机用来处理本地的小型重复任务，不需要面向对象，反而内存过小导致程序员要认真考虑分配，所以是C；然后硬件发展，同时也有了一些大型任务要求不同端口交互，就有了C++；硬件继续发展，内存管理越来越不重要，反而更加追求代码的简洁，然后就有了python（一些感想，忽略即可）.

1.4 条件语句 if

> If statement

Selection statement for 1 or more choices

```
if choice == 1:
    print ('You choose Cola')
    out = 'coke'
elif choice == 2:
    print ('You choose Lemon tea')
    out = 'Lemon Tea'
elif choice == 3:
    print ('You choose Orange juice')
    out = 'Orange Juice'
else:
    print ('Invalid choice!')
    print ('choose again')
```

> 注意条件不需要小括号
>
> 注意每个 Condition 后要加 `:`
>
> 不需要花括号，但要注意缩进，不能省略
>
> 缩进可以是任意大于 0 的宽度，但同一个代码块的不同行，缩进要等宽

```
choice = 4
if choice == 1:
    print ('You choose Cola')
    out = 'coke'
elif choice == 2:
    print ('You choose Lemon tea')
    out = 'Lemon Tea'
elif choice == 3:
 print ('You choose Orange juice')
 out = 'Orange Juice'
else:
                print ('Invalid choice!')
                print ('choose again')
```

这样也能正常跑，控制台输出：

```
Invalid choice!
choose again
```

进阶：Conditional Expressions

```
var = true_value if condition else false_value
```

which is equivalent to:

```
if condition:
var = true_value
else:
var = false_value
```

For example:

```
a, b = 10, 20 # Multiple assignments
min = a if a < b else b
print(min) # output 10
```

1.5 循环语句 for, while

while 循环

```
a, b = 0, 1
while b < 10:
    print (b)
    a, b = b, a+b
print ('END')
```

```
1
1
2
3
5
8
END
```

for 循环

```
for x in range(0, 3) : # 左闭右开
    print (x)
```

```
0
1
2
```

1.6 格式化，占位符

> Output Formatting

```
print('%s %s' % ('one', 'two')) # old
print('{} {}'.format('one', 'two')) # new

print('%d %d' % (1, 2))
print('{} {}'.format(1, 2))
```

```
one two
one two
1 2
1 2
```

> 两种都能用

WIth new style formatting, it is possible to give placeholders an explicit positional index.

```
print('{1} {0}'.format('one', 'two'))
```

```
two one
```

占位符的花式玩法

```
txt = "For only {price:.2f} dollars!"
print(txt.format(price = 32.1))
```

```
For only 32.10 dollars!
```

```
txt1 = "My name is {fname}, I'am {age}".format(fname = "Noelle", age = 2.5)
txt2 = "My name is {0}, I'am {1}".format("Noelle", 2.5)
txt3 = "My name is {}, I'am {}".format("Noelle", 2.5)
print(txt1)
print(txt2)
print(txt3)
```

```
My name is Noelle, I'am 2.5
My name is Noelle, I'am 2.5
My name is Noelle, I'am 2.5
```

Lecture 2: Lists and Tuples

> 列表和元组

2.1 Strings

> Python 中 `String` 以 a sequence of characters 的方式储存.
>
> 注意：`str` 和 `list` 是两种不同的数据类型，虽然它们的索引方式很相似.
>
> 有多种方式可以把 `str` 转为 `list`，如 `list(str)`（将 `str` 中的每个字符分别作为列表中的元素），`str.split(sep)` 等.
>
> Use single quote `'` or double quote `"` to delimit.
>
> `\` : escape char（转义字符）

Escape character	Printed as
`\'`	Single quote
`\"`	Double quote
`\t`	Tab
`\n`	Newline
`\\`	Backslash

2.1.1 拼接操作

除了加号 concatenate 之外也可以乘号 repeat .

```
>>> word = 'CUHK'+ 'super'
>>> word
'CUHKsuper'
>>> '[' + word*3 + ']'
'[CUHKsuperCUHKsuperCUHKsuper]'
```

2.1.2 访问元素

Access can be in subscripted notation

> 可以用下标访问（类似 C 的数组，从 0 开始）

```
>>> word = 'CUHK'+ 'super'
>>> word[2]
'H'
```

2.1.3 访问切片

substrings can be specified with the slice notation

> 切片表示法，左闭右开

```
>>> word = 'CUHK'+ 'super'
>>> word[0:2]
'CU'
>>> word[:2]
'CU'
>>> word[1:]
'UHKsuper'
```

> 注意第三个，如果没标 end，默认延续到序列末尾，并且包括最后一个元素.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds.

> 对于范围内的非负索引，切片长度是索引之差.

***关于负索引、超范围索引和可能的异常**

> 课件没有，仅作补充.

① 负数索引：从末尾向开头反向索引.

> -1 表示最后一个元素，-2 表示倒数第二个元素.
>
> 注意 -0 也被视为 0，而不是最后一个元素的下一个位置.

```
>>> word = 'CUHK'+ 'super'
>>> word[-3:]
'per'
>>> word[-4:-1]
'upe'
>>> word[1:-1]
'UHKsupe'
```

> 注意仍然是左闭右开.

② 索引超出范围，自动调整，尽量减少丢失信息.

> 正索引超出，自动调整为最后一个元素的**下一个位置**.
>
> 负索引超出，自动调整为第一个元素的位置.
>
> 二者不同是因为区间左闭右开，导致不对称.
>
> 缺点：不抛出异常，可能影响 debug 效率.
>
> 注意：只是切片的索引超范围不抛 error，单个索引会抛.

③ 反向切片：若 start 索引比 end 索引大，且步长为负，则切片从右往左.

```
>>> word = 'CUHK'+ 'super'
>>> word[-1:0:-2]
'rpsH'
```

> 第三个参数是步长，注意仍然左闭右开，字符 C 没有输出.
>
> 若 start 索引比 end 索引大，且步长为正，则输出空序列. 因为切片方向与步长方向不一致.

④ 切片操作可能抛出的异常

ValueError：步长为 0（没有意义）.

```
ValueError: slice step cannot be zero
```

TypeError：有两种情况.

情况1：切片的 `start`、`end`、`step` 参数必须是整数或可以隐式转换为整数的类型（如 `None`）. 如果传入了非整数类型（如字符串、浮点数等），Python 会抛出 `TypeError`.

```
TypeError: slice indices must be integers or None or have an __index__ method
```

情况2：对不支持切片的对象（如整数）尝试切片，Python 会抛出 `TypeError`.

```
TypeError: 'int' object is not subscriptable
```

还有一些比较少见的 error，如序列过大导致的内存溢出 `MemoryError`，索引过大溢出 python 可处理整数范围的 `OverflowError`，以及对非字符串的自定义对象切片但类中没有实现方法，可能抛出 `AttributeError`.

注意：字符串不可修改

Immutable – can't be modified once created

```
>>> word = 'CUHK'+ 'super'
>>> word[0] = 2
```

异常信息：

```
TypeError: 'str' object does not support item assignment
```

但是，我们可以创建新的

```
>>> word = 'CUHK'+ 'super'
>>> word[0:3]+ " course"
'CUH course'
```

> 注意，如果重名会覆盖掉原来的.

2.1.4 分割操作

> Splitting Strings

`str.split()` 方法用于分割字符串，如果没有提供分隔符参数，默认使用空白字符（空格、制表符等）作为分割符，并自动忽略多余的空白字符.

```
message = "CSCI2040 is really boring !!!, really very boring."
words = message.split(' ') # split uses ''(or space) as separator
print(words)
words = message.split('really') # split uses 'really'as separator
print(words)
```

输出如下：

```
['CSCI2040', 'is', 'really', 'boring', '!!!,', 'really', 'very', 'boring.']
['CSCI2040 is ', ' boring !!!, ', ' very boring.']
```

不传参，默认空白字符为分隔符，并忽略多余空格.

> 意思就是不传参的话，切出来的 token 里不会有空格.

```
message = "CSCI2040   is   really   boring   !!!,   really  very  boring."
words = message.split() # split uses ''(or space) as separator
print(words)
words = message.split('really') # split uses 'really'as separator
print(words)
```

输出如下：

```
['CSCI2040', 'is', 'really', 'boring', '!!!,', 'really', 'very', 'boring.']
['CSCI2040   is   ', '   boring   !!!,   ', '  very  boring.']
```

注意，不允许使用空字符串 `''` 作为分隔符.

> 没有意义.

```
ValueError: empty separator
```

2.1.5 String Functions

> Note the use of `.` to call the string function.

Operation	Description	Example
`s.capitalize()`	Capitalize the first character of `s`	"hello world".capitalize() → 'Hello world'
`s.capwords()`	Capitalize the first letter of each word in `s`	"hello world".title() → 'Hello World'
`s.count(sub)`	Count number of occurrences of `sub` in `s`	"banana".count('a') → 3
`s.find(sub)`	Find the first index of `sub` in `s`, or `-1` if not found	"hello world".find('o') → 4 "hello world".find('x') → -1
`s.index(sub)`	Find the first index of `sub` in `s`, or raise a `ValueError` if not found	"hello world".index('o') → 4 "hello world".index('xyz') → ValueError: substring not found
`s.rfind(sub)`	Find the last index of `sub` in `s`, or `-1` if not found	"hello world".rfind('o') → 7 "hello world".rfind('xyz') → -1
`s.rindex(sub)`	Find the last index of `sub` in `s`, or raise a `ValueError` if not found	"hello world".rindex('o') → 7 "hello world".rindex('xyz') → ValueError: substring not found
`s.lower()`	Convert `s` to lowercase	"HELLO".lower() → 'hello'
`s.split()`	Return a list of words in `s`	"hello world".split() → ['hello', 'world']
`s.join(lst)`	Join a list of words into a single string with `s` as the separator	" ".join(['hello', 'world']) → 'hello world'
`s.strip()`	Strip leading/trailing white space from `s`	" hello world ".strip() → 'hello world'
`s.upper()`	Convert `s` to uppercase	"hello".upper() → 'HELLO'
`s.replace(old, new)`	Replace all instances of `old` with `new` in `s`	"hello world".replace('world', 'Python') → 'hello Python'

`.strip()`：strip 是去除的意思，这个方法由于去除字符串开头和结尾的空白（中间的不影响）

> 考试需要使用应该会提供声明.
>
> 开头结尾所有空白都被去除，即使有多个单位空字符.

2.2 Lists

> 类似 C 的数组

Python 的列表很强大，可以容纳不同的数据类型.

2.2.1 主要操作和方法

> 以下流程覆盖了主要操作逻辑，过一遍即可.

```
>>> squares = [1, 4, 9, 16, 25] # 初始化
>>> squares[1] # 单索引访问元素（和字符串相似）
4
>>> squares[1] = 'Hello' # 可修改
>>> squares # 支持不同数据类型
[1, 'Hello', 9, 16, 25]
>>> squares[-1] # 反向索引（和字符串相似）
25
>>> squares[-4] = 4 # 反向索引也可修改
>>> squares
[1, 4, 9, 16, 25]
>>> squares[-3:] # 访问切片（和字符串相似）
[9, 16, 25]
>>> squares[:3] # 访问切片（左闭右开）
[1, 4, 9]
>>> squares[:] # 访问切片（全覆盖）
[1, 4, 9, 16, 25]
>>> squares = squares + [36, 49, 64, 81, 100] # Support concatenation 加号拼接
>>> squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> squares.append(121) # 内置方法，末尾追加（参数任意类型，不能为空，可以为 None）
>>> squares[10]
121
>>> squares *= 3 # 支持星号拼接
>>> squares
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100,
121, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # 初始化另一个 list
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters # 切片也可修改（左闭右开）
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> letters[2:5]=[] # 利用切片修改进行移除
>>> letters # 切片修改的数量不需要对应
['a', 'b', 'f', 'g']
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # 初始化（重名，覆盖掉原来的）
>>> len(letters) # 返回 list 长度
7
>>> letters[7] # 单索引超范围，抛 IndexError
Traceback (most recent call last):
  File "<pyshell#87>", line 1, in <module>
    letters[7]
IndexError: list index out of range
>>> letters[0:7] # 切片索引超范围，不抛 error
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

> 注意点：
>
> 1. the index of the list starts from 0
> 2. 区间都是左闭右开
> 3. list 和 string 的区别：列表可变，string 不可变；列表支持不同数据类型，string 只支持字符；列表可修改，有许多修改本身的内置 method，string 的内置方法只能返回新字符串.
> 4. list 和 string 的相同点：相同的索引模式（同样的单索引不能超范围，同样的切片索引可以超，同样的负数索引模式）

2.2.2 for 循环遍历

> 可以利用循环操作遍历列表（字符串也可以）

```
a = ['HKU', 'CUHK', 'UST']
for x in a:
    print (x, len(x))
```

输出如下：

```
HKU 3
CUHK 4
UST 3
```

`range()` 函数

> 返回一个 range 对象，range 是 python 的一个数据类型（类）.

语法

```
range(stop)
range(start, stop)
range(start, stop, step)
```

> start：起始值，默认是 0.
>
> stop：结束值，不包含 stop，因为左闭右开.
>
> step：步长，默认是 1.
>
> 只有 stop 参数必填，其他不填就用默认值.
>
> 只输入两个参数时，第一个是 start，第二个是 stop，左闭右开.

```
r = range(1, 10, 2)  # 生成从 1 开始，到 9 的奇数序列
print(type(r))  # 输出: <class 'range'>
print(list(r))  # 输出: [1, 3, 5, 7, 9]
```

for 循环遍历 range 对象

```
for i in range(5):
    print(i)
```

输出如下：

```
0
1
2
3
4
```

range 对象支持索引

```
r = range(10, 20)
print(r[0])  # 输出: 10
print(r[5])  # 输出: 15
```

注意：range对象不可变，类似 Tuple，不能修改元素（想犯这个错也挺难的）.

***range 对象的其他特性**

> （不考）

懒惰生成：range 不会立即生成整个序列，而是按需生成每个值，节省内存.

支持迭代：简单来说就是可以被 for 循环遍历.

***可迭代对象**

> （不考）

可迭代（Iterable）是 Python 中的一个概念，指的是一个对象能够返回其元素的一个迭代器，从而可以逐一访问每个元素.

> 简单来说，**可迭代对象**是可以用在 `for` 循环中遍历的对象，或者可以被传递给像 `sum()`、`min()`、`max()` 等接受可迭代对象的函数.

在 Python 中，**可迭代对象**是实现了 `__iter__()` 方法或实现了 `__getitem__()` 方法的对象.

常见可迭代对象包括：

- **序列类型**：如 `list`（列表）、`tuple`（元组）、`str`（字符串）、`range` 对象等.
- **集合类型**：如 `set`（集合）、`dict`（字典）.
- **生成器**：生成器函数返回的对象也是可迭代的.

示例：常见可迭代对象

```
# 列表是可迭代对象
my_list = [1, 2, 3, 4]
for item in my_list:
    print(item)

# 字符串是可迭代对象
my_str = "hello"
for char in my_str:
    print(char)

# range 对象是可迭代对象
for i in range(3):
    print(i)
```

输出：

```
1
2
3
4

h
e
l
l
o
```

```
0
1
2
```

***迭代器**

可迭代对象本身不是迭代器，但可通过调用 `iter()` 函数将其转换成**迭代器**.

迭代器是一个实现了 `__next__()` 方法的对象，它能够逐一返回可迭代对象中的元素.

示例：将可迭代对象转换为迭代器

```python
my_list = [1, 2, 3]
my_iter = iter(my_list)  # 将列表转换为迭代器

print(next(my_iter))  # 输出: 1
print(next(my_iter))  # 输出: 2
print(next(my_iter))  # 输出: 3
```

在这个例子中，`my_list` 是一个可迭代对象，通过调用 `iter(my_list)`，我们得到了一个迭代器 `my_iter`. 然后通过 `next()` 函数，我们逐个获取列表中的元素.

3. **如何判断对象是否可迭代**

可以使用 Python 内置模块 `collections.abc` 中的 `Iterable` 来判断一个对象是否可迭代.

示例：判断是否可迭代

```python
from collections.abc import Iterable

# 列表是可迭代对象
print(isinstance([1, 2, 3], Iterable))  # 输出: True

# 字符串是可迭代对象
print(isinstance("hello", Iterable))    # 输出: True

# 整数不是可迭代对象
print(isinstance(10, Iterable))         # 输出: False
```

可迭代对象的作用

可迭代对象允许我们使用 `for` **循环** 或 `while` **循环** 来遍历其内容，使代码更加简洁和易于维护.

常见的 Python 内置函数，如 `sum()`、`max()`、`min()`、`sorted()` 等，接受可迭代对象作为输入.

示例：使用 `sum()` 和 `sorted()` 处理可迭代对象

```python
numbers = [4, 1, 3, 2]

# sum() 计算可迭代对象的总和
print(sum(numbers))  # 输出: 10

# sorted() 对可迭代对象进行排序
print(sorted(numbers))  # 输出: [1, 2, 3, 4]
```

*生成器

生成器也是一种可迭代对象，但它是惰性计算的（即按需生成元素）. 生成器通过 `yield` 语句生成元素，避免一次性创建整个序列，从而节省内存.

示例：生成器函数

```python
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()

# 生成器是可迭代对象
for num in gen:
    print(num)
```

输出：

```
1
2
3
```

> 注意，range 和生成器类似，都有懒惰生成，但 range 不是生成器，区别在于：
>
> ① 生成器通过 `yield` 关键字逐步生成值，每次调用时才会计算下一个值. range 虽然也有这个特性，但是它不是 `yield` 定义，而是通过 C 语言实现的一个类.
>
> ② range 对象支持索引，生成器不支持.
>
> ③ range 对象可多次迭代（重复使用），生成器只能迭代一次.

总结

① **可迭代对象**是可以使用 `for` 循环遍历的对象，常见的有 `list`、`tuple`、`str`、`dict`、`set` 等.

② 可迭代对象实现了 `__iter__()` 或 `__getitem__()` 方法，返回一个迭代器.

③ **迭代器**是通过 `iter()` 转换来的对象，它实现了 `__next__()` 方法，可以逐个返回元素.

④ 可迭代对象使得代码更加简洁和易维护，且可以与许多 Python 内置函数配合使用.

所以，**可迭代**意味着一个对象可以被逐个访问其元素，通常通过 `for` 循环或迭代器.

*迭代器和生成器的区别

> 待补充.

`range()` + `len()` 遍历 list

We retrieve the item through their index here.

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in (range(len(a))):
    print (i, a[i])
```

输出如下:

```
0 Mary
1 had
2 a
3 little
4 lamb
```

`enumerate(list)` 实现

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i, value in enumerate(a):
    print(i, value)
```

输出如下:

```
0 Mary
1 had
2 a
3 little
4 lamb
```

Another Example:

```
cse_teachers = ['john c. s. lui', 'patrick p. c. lee', 'james cheng', 'wei Meng']
print ("------- use enumerate -----------")
for index, teacher in enumerate(cse_teachers):
    position = str(index) # 下面要用到加号拼接，加号不能拼接int和str
    print("Position: " + position + " Teacher: " + teacher.title())
```

输出如下：

```
------- use enumerate -----------
Position: 0 Teacher: John C. S. Lui
Position: 1 Teacher: Patrick P. C. Lee
Position: 2 Teacher: James Cheng
Position: 3 Teacher: Wei Meng
```

`enumerate(a)` 返回一个迭代器，该迭代器在每次迭代时返回一个包含索引和对应元素的元组 `(i, value)` .

> 返回元组主要因为它不可变. 索引和对应元素是一对紧密关联的值，不应该被修改，返回元组保证安全性.

使用 `for i, value in enumerate(a)`，就可以直接解包元组，得到索引 `i` 和对应的元素 `value` .

> enumerate 是枚举的意思.

解包并不是元组的专利，列表也可以解包，侧面说明返回元组的主要原因是安全考量.

解包列表 Example：

```
def index_value_generator(my_list):
    for index in range(len(my_list)):
        yield [index, my_list[index]]  # 每次返回一个列表 [index, value]

# 使用生成器进行迭代
my_list = ['a', 'b', 'c']

for item in index_value_generator(my_list):
    print(item)  # 每次输出一个列表 [index, value]

for index, value in index_value_generator(my_list):
    print(index, value)  # 解包列表
```

输出如下：

```
[0, 'a']
[1, 'b']
[2, 'c']
0 a
1 b
2 c
```

注意，两个循环创建了两次生成器.

不能写成这样：

```
def index_value_generator(my_list):
    for index in range(len(my_list)):
```

```
        yield [index, my_list[index]]  # 每次返回一个列表 [index, value]

# 使用生成器进行迭代
my_list = ['a', 'b', 'c']

generator = index_value_generator(my_list)

for item in generator:
    print(item)  # 每次输出一个列表 [index, value]

for index, value in generator:
    print(index, value)  # 解包列表
```

> 因为生成器只能迭代一个周期.

错误输出如下:

```
[0, 'a']
[1, 'b']
[2, 'c']
```

直接迭代 list

list 是可迭代对象，如果只需要遍历元素而不关心索引，可以直接迭代列表:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for value in a:
    print(value)
```

输出如下:

```
Mary
had
a
little
lamb
```

Another Example:

```
cse_teachers = ['john c. s. lui', 'patrick p. c. lee', 'james cheng', 'wei Meng']
print ("------- use list.index(value) -----------")
for teacher in cse_teachers:
    print("Position:", cse_teachers.index(teacher), "Teacher: " + teacher)
```

输出如下:

```
------- use list.index(value) -----------
Position: 0 Teacher: john c. s. lui
Position: 1 Teacher: patrick p. c. lee
Position: 2 Teacher: james cheng
Position: 3 Teacher: wei Meng
```

> 这里有很大风险，`list.index(x)` 只会返回第一个匹配元素的索引，如果有重复元素，结果就会出问题. 因此，对于同时输出索引和元素的迭代，我们优先考虑 `enumerate(list)` 实现，如果非要使用 `list.index(x)` ，必须保证列表没有重复元素.

2.2.3 `in` 检查

> `in` 运算符用于检查一个值是否存在于容器（如列表、字符串等）中，并返回一个布尔值：`True` 或 `False` .

语法

```
element in list
```

> 当 `element` 在 `list` 中时，`in` 运算符返回 `True` ，否则返回 `False` .
>
> 大小写敏感，需要完全匹配.
>
> 返回的是布尔值，可以被 print.

```
cse_teachers = ['john c. s. lui', 'patrick p. c. lee', 'y.p.chui', 's.h.or']
print ("Is 'john c. s. lui'in the list?", 'john c. s. lui' in cse_teachers)
print ("Is 'John C. S. Lui'in the list?", 'John C. S. Lui' in cse_teachers)
```

输出如下：

```
Is 'john c. s. lui'in the list? True
Is 'John C. S. Lui'in the list? False
```

> Case sensitive

2.2.4 List Operations

Operation	Description	Example
`s.append(x)`	Appends element `x` to `s`	`>>> s = [1, 2]; s.append(3); s` `[1, 2, 3]`
`s.extend(ls)`	Appends list `ls` with `s`	`>>> s = [1, 2]; s.extend([3, 4]); s` `[1, 2, 3, 4]`
`s.count(x)`	Counts number of occurrences of `x` in `s`	`>>> s = [1, 2, 2]; s.count(2)` `2`
`s.index(x)`	Returns index of **first occurrence** of `x`	`>>> s = [1, 2, 3]; s.index(2)` `1`

`s.pop()`	Returns and removes last element from `s`	`>>> s = [1, 2, 3]; s.pop()`
		`3`
		`>>> s`
		`[1, 2]`

`s.pop(i)`	Returns and removes element at index `i`	`>>> s = [1, 2, 3]; s.pop(1)`
		`2`
		`>>> s`
		`[1, 3]`

`s.remove(x)`	Searches for `x` and removes it from `s`	`>>> s = [1, 2, 3]; s.remove(2); s` `[1, 3]`
`s.reverse()`	Reverses elements of `s` in place	`>>> s = [1, 2, 3]; s.reverse(); s` `[3, 2, 1]`
`s.sort()`	Sorts elements of `s` into ascending order	`>>> s = [3, 1, 2]; s.sort(); s` `[1, 2, 3]`
`s.insert(i, x)`	Inserts `x` at location `i`	`>>> s = [1, 3]; s.insert(1, 2); s` `[1, 2, 3]`

> 注意：
>
> `s.insert(i, x)`：将元素 `x` 插入到索引 `i` 的位置，原先在位置 `i` 的元素及后续所有元素**顺次向右移动**，以腾出位置给 `x` .

2.3 Tuples

> 元组

与列表相似，但使用小括号.

不可变，这是与列表的主要区别.

列表的非可变操作可以应用于元组.

2.3.1 主要操作

```
>>> tup = (1,7,3,1,7,3)
>>> 3 in tup
True
>>> list(tup)
[1, 7, 3, 1, 7, 3]
>>> tuple(['a', 'b', 'c'])
('a', 'b', 'c')
```

*注意：对于元组，初始化**单元素元组**需要在元素后加一个逗号来区分，例如 `(1,)` . 否则 `(1)` 会被视为普通的括号运算.

> 不考

```
>>> a = (1)
>>> a
1
>>> a=(1,)
>>> a
(1,)
```

2.3.2 隐式创建

Comma operator implicitly creates a tuple.

```
>>> 'a','b','c'
('a', 'b', 'c')
```

2.3.3 元组解包

Python 允许通过解包（也叫**拆包**）的方式，将一个元组或其他可迭代对象的多个元素直接赋值给多个变量.

Application in function returning more than 1 result.

```
def minAndMax( info ):
    return (min(info), max(info))
>>> x, y = minAndMax('abcd')
>>> x
'a'
>>> y
'd'
```

2.4 实战：密码强度检测

What good list & strings brings?

Consider the following problem:

Write a program to check whether an input string is a strong password, which has to contain:

① lower case letters,

② upper case letters,

③ digits,

④ special characters, and

⑤ the length has to be no shorter than 12.

C 语言

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

// 函数用于检查密码是否强
int isStrongPassword(char password[]) {
    int hasLower = 0, hasUpper = 0, hasDigit = 0, hasSpecial = 0;
    int length = strlen(password);

    // 如果密码长度小于12，直接返回0（密码不强）
    if (length < 12) {
        return 0;
    }

    // 遍历密码中的字符
    for (int i = 0; i < length; i++) {
```

```
        if (islower(password[i])) {
            hasLower = 1;
        } else if (isupper(password[i])) {
            hasUpper = 1;
        } else if (isdigit(password[i])) {
            hasDigit = 1;
        } else if (ispunct(password[i])) {  // 检查是否为特殊字符
            hasSpecial = 1;
        }
    }

    // 检查是否满足所有条件
    return hasLower && hasUpper && hasDigit && hasSpecial;
}

int main() {
    char password[100];

    printf("请输入密码: ");
    scanf("%s", password);

    if (isStrongPassword(password)) {
        printf("这是一个强密码!\n");
    } else {
        printf("密码不符合强密码要求。\n");
    }

    return 0;
}
```

Python

```
import string

def is_strong_password(password):
    # 检查长度
    if len(password) < 12:
        return False

    # 初始化标志变量
    has_lower = has_upper = has_digit = has_special = False

    # 遍历密码字符
    for char in password:
        if char.islower():
            has_lower = True
        elif char.isupper():
            has_upper = True
        elif char.isdigit():
            has_digit = True
        elif char in string.punctuation:  # 检查特殊字符
            has_special = True
```

```
    # 检查是否满足所有条件
    return has_lower and has_upper and has_digit and has_special

# 获取用户输入
password = input("请输入密码: ")

# 检查密码强度
if is_strong_password(password):
    print("这是一个强密码！")
else:
    print("密码不符合强密码要求。")
```

区别：

C 考虑的更底层，遍历密码的参数细节都要考虑，对程序员要求比较高

Python 更强调整体结构和方法调用，对具体参数要求不高，不烧脑

*2.5 比较 Strings, Lists, Tuples

> 自己总结的，不考，但整体思想会渗透到考题.

① 可变性

`str`：不可变.

`list`：可变（修改、添加、删除）.

`tuple`：不可变.

② 语法

`str`：单引号 `'` 或双引号 `"` 括住字符序列.

`list`：方括号 `[]` 括住，元素用逗号分隔.

`tuple`：圆括号 `()` 括住，元素用逗号分隔.

③ 元素类型

`str`：只能包含字符（本质上是单个字符的序列）.

`list`：可包含任意类型的元素.

`tuple`：可包含任意类型的元素.

④ 内存与性能

`str`：不可变，内存占用小，访问速度快.

`list`：可变，需要更多内存以支持动态修改，性能稍慢.

`tuple`：不可变，内存占用小，访问速度快.

⑤ 操作

操作	`str`	`list`	`tuple`
访问元素	允许	允许	允许
修改元素	不允许	允许	不允许
添加元素	不允许	允许	不允许
删除元素	不允许	允许	不允许
切片	允许	允许	允许
拼接	允许	允许	允许
迭代	允许	允许	允许
长度 `len()`	允许	允许	允许
是否可变	不可变	可变	不可变

⑥ 使用场景

`str`：存储和操作**文本**数据.

`list`：需要动态修改、扩展的有序数据集合.

`tuple`：存储不可变、有序的数据集合，尤其适用于表示固定的记录、数据对等不需要修改的场景.

> 例如列表索引和对应的元素值，在迭代的时候应该一一对应，因此 `enumerate(list)` 返回的迭代器在每次迭代时返回的是一个包含索引和对应元素的元组 `(index, value)`，保证安全性.

Lecture 3: Functions

3.1 导入模块

> Import module

3.2 定义和调用函数

> Define & Invoking functions

3.3 传参

> Parameters passing

3.4 返回值

> Return value

3.5 列表推导式

> List Comprehensions

列表推导式是 `Python` 中一种简洁的创建列表方式. 它允许**在一行代码中通过表达式和循环快速生成列表**，而不需要使用传统的 `for` 循环和 `append()` 方法. 不仅简洁，而且执行速度通常比传统循环更快.

3.5.1 语法

```
[expression for item in iterable if condition]
```

`expression` ： 生成列表元素的表达式. 可以是简单的值，也可以是基于 `item` 的操作.

`item` ： 从 `iterable` 中遍历的元素.

`iterable` ： 可迭代对象，如列表，字符串，元组等.

> 见 `2.2.2 - 可迭代对象` .

`condition` ： 可选，用于筛选元素的表达式.

3.5.2 示例

① 基本用法

生成一个包含从 0 到 9 的平方数的列表.

```
squares = [x**2 for x in range(10)]
print(squares)
```

> 注意左闭（默认 0）右开（10 取不到）
>
> `**` 幂运算符，左边底数，右边指数. 这里是 x^2.

输出如下：

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

② 带条件的列表推导式

生成一个列表，包含 0 到 9 中偶数的平方.

> 0 到 9 不全是偶数，需要引入 if 判断.

```
even_squares = [x**2 for x in range(10) if x % 2 == 0]
print(even_squares)
```

输出如下：

```
[0, 4, 16, 36, 64]
```

③ 调用函数的列表推导式

给定一个字符串列表，将每个元素转小写.

> 可以在表达式中调用函数.

```
words = ['Hello', 'WORLD', 'Python']  # 给定字符串列表
lower_words = [word.lower() for word in words]
print(lower_words)
```

> 关于 `.lower()` 函数，见 `2.1.5 String Functions`.

输出如下：

```
['hello', 'world', 'python']
```

④ 嵌套列表推导式

给定一个 2D 矩阵，生成一个列表，包含它所有元素.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in matrix for num in row]
print(flattened)
```

输出如下：

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

⑤ 带条件的嵌套列表推导式

给定一个 2D 矩阵，生成一个列表，包含所有偶元素.

> 在 ④ 的基础上过滤掉奇数.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
even_nums = [num for row in matrix for num in row if num % 2 == 0]
print(even_nums)
```

输出如下：

```
[2, 4, 6, 8]
```

（有时间的话，把lab4的题补充进来.）

⑥

⑦

⑧

⑨

⑩

3.5.3 优点

① 简洁：一行代码，简洁易读.

② 快速：底层优化，减少函数调用开销，比 for + append 快.

可以对比一下，创建包含 1 到 10 平方的列表：

for + append

```
squares = [] # 还要初始化一个空列表
for i in range(1, 11):
    squares.append(i ** 2)
print(squares)  # 输出：[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

> 关于 `list.append(x)` ，见 `2.2.4 List Operations` .

List Comprehensions

```
squares = [i ** 2 for i in range(1, 11)]
print(squares)  # 输出：[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

3.5.4 应用

① 从一个列表生成新列表

例：将给定列表中所有元素加倍.

```
numbers = [1, 2, 3, 4, 5]
doubled = [x * 2 for x in numbers]
print(doubled)
```

```
[2, 4, 6, 8, 10]
```

② 筛选元素

例：从给定列表中筛选出所有负数.

```
numbers = [-5, -2, 0, 3, 7]
negatives = [x for x in numbers if x < 0]
print(negatives)
```

```
[-5, -2]
```

③ 字符处理

例：提取给定字符串中所有字母字符.

```
sentence = "Hello, World!"
letters = [char for char in sentence if char.isalpha()]
print(letters)
```

```
['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']
```

> 注意单引号. `print(容器)` 会调用容器内部元素的 `repr()` 方法来显示元素，和直接 `print(元素)` 略有不同. 主要体现在字符和字符串类型的元素，直接 `print(char/str元素)` 不带引号，`print(包含它的容器)` 带引号.

④ 嵌套列表推导式

例：创建九九乘法表，2D 矩阵，元素是乘法结果.

```
multiplication_table = [[i * j for j in range(1, 10)] for i in range(1, 10)]
print(multiplication_table)
```

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

3.6 作用域

> Scope

3.7 Lambda 函数

> Lambda function

① 可在一行代码内定义.

② 没有函数名，被称为匿名函数.

③ 语法简洁，但功能仅限于简单表达式.

④ 短期使用、简单场景.

3.7.1 语法

```
lambda arguments: expression
```

`lambda`：定义 Lambda 函数的关键字.

`arguments`：函数的参数，可有多个，用逗号隔开.

`expression`：函数体，单个表达式，不能包含复杂语句. 表达式的结果即该匿名函数的返回值.

3.7.2 示例

① 单参数

例：定义 Lambda 函数，计算一个数的平方.

```
square = lambda x: x ** 2
print(square(4))  # 输出: 16
```

> `=` 把匿名函数对象的**引用**传给 `square` 变量，并不代表该匿名函数有了名字 `square`，`square` 只是存了函数对象的引用.
>
> 引用，类似 C 的地址，square 类似指向该函数地址的指针，只是 python 自动帮你解引用. 当然底层实现和指针有许多不同，暂时先这么理解.
>
> 同为面向对象语言，Python 和 Java 关于引用有诸多相似之处，例如垃圾回收机制.

② 多参数

例：定义 Lambda 函数，计算两数和.

```
add = lambda x, y: x + y
print(add(3, 5))  # 输出: 8
```

③ 不含参

```
greet = lambda: "Hello, World!"
print(greet())  # 输出: Hello, World!
```

> 恒定的返回值.

3.7.3 应用

① 与内置函数结合

(1) `map()`

`map()` 函数将一个函数作用到一个可迭代对象的每个元素上，返回一个迭代器.

> 迭代器懒惰求值，要访问或打印，通常需要转为列表（或元组，集合等）.
>
> 关于迭代器，见 `2.2.2 for 循环遍历 - range() 函数 - *迭代器` .

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers)) # list()遍历迭代器，将元素存进列表
print(squared_numbers)  # 输出: [1, 4, 9, 16, 25]
```

(2) `filter()`

`filter()` 函数用一个返回布尔值的函数对一个可迭代对象的元素逐个检查，过滤掉不满足条件的元素，返回过滤后的迭代器.

> 返回 True 保留，返回 False 过滤掉.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # 输出: [2, 4, 6, 8, 10]
```

(3) `sorted()`

`sorted()` 函数对列表排序，返回新列表. Lambda 函数可用于指定排序规则.

```
points = [(1, 2), (3, 1), (5, 0), (2, 4)]
sorted_points = sorted(points, key=lambda x: x[1])
print(sorted_points)  # 输出: [(5, 0), (3, 1), (1, 2), (2, 4)]
```

> 本例中，`lambda x: x[1]` 定义了一个匿名函数，指定 `sorted()` 根据每个元组第二个元素排升序.

`sorted()` 语法

```
sorted(iterable, *, key=None, reverse=False)
```

> 待补充.

② 与非内置函数结合

例 1. 自定义函数

```
def apply_operation(x, y, operation):
    return operation(x, y)

# 使用 lambda 函数作为参数
result = apply_operation(5, 3, lambda a, b: a * b)
print(result)  # 输出: 15
```

例 2. `reduce()` 函数

```
from functools import reduce

# 列表中的数字
numbers = [1, 2, 3, 4, 5]

# 使用 reduce 和 lambda 来计算乘积
result = reduce(lambda x, y: x * y, numbers)

print(result)  # 输出: 120
```

③ 与列表推导式结合

例：对给定列表的每个元素应用一个 Lambda 函数.

```
numbers = [1, 2, 3, 4, 5]
doubled = [(lambda x: x * 2)(x) for x in numbers]
print(doubled)  # 输出: [2, 4, 6, 8, 10]
```

> 关于列表推导式，见 `3.5 列表推导式` .

Lecture 4: File Operations, Pickle & Dictionary

4.1 `input()`

程序执行到 `input()` 时，会暂停，等待用户在控制台中输入内容，将输入内容作为**字符串**返回.

> 注意，如果不进行类型转换，即使键盘输入数字，得到的也是字符串.

4.1.1 语法

```
input(prompt)
```

> `prompt`：可选. 一个字符串，显示在输入提示符前. 不影响用户输入的值，只是提示用户输入.

4.1.2 示例

① 获取字符串输入

```
# 获取用户输入的名字
name = input("请输入你的名字: ")
print(f"你好, {name}!")
```

> `input()` 本身是函数，要把它的返回值赋给一个变量（比如这里的 `name`），才能把信息保存下来.

② 获取数字输入

```
# 获取用户输入的数字，并转换为整数
age = int(input("请输入你的年龄: "))
print(f"你已经 {age} 岁了。")
```

> `int()`，`float()` 用于类型转换，分别将数据转为整数和浮点数.
>
> `int()` 接受的参数类型：表示整数的字符串、浮点数（截取整数部分）、布尔值（返回 1 或 0）、实现了 `__int__()` 方法的对象. 它还可以处理不同进制的数字字符串.
>
> `float()` 接受的参数类型：字符串（可以是科学计数法字符串）、整数、布尔值、实现了 `__float__()` 方法的对象.
>
> 两者都不能处理无法转为数字的字符串，如果传入无效字符串，会抛出 `ValueError`.
>
> `int()` 可以处理浮点数，但是不能处理表示浮点数的字符串，会抛出 `ValueError`.

```
>>> int('36')
36
>>> int('12 int')
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int('12 int')
ValueError: invalid literal for int() with base 10: '12 int'
```

```
>>> int('12.34')
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    int('12.34')
ValueError: invalid literal for int() with base 10: '12.34'
>>> int('12 ')
12
>>> int('1010', 2) # 第二个参数是进制
10
>>> float('36')
36.0
```

> 若字符串表示整数，转 `float` 保留 1 位小数，因为 Python 的浮点数输出通常只显示必要的位数（有 1 位足以说明它是浮点，再加 0 没有意义）.

如果要自定义小数位数，可以用格式化字符串：

```
# 字符串表示的整数
s = "123"

# 转换为浮点数，并保留两位小数
formatted_number = f"{int(s):.2f}"
print(formatted_number)  # 输出: 123.00

# 保留三位小数
formatted_number = f"{int(s):.3f}"
print(formatted_number)  # 输出: 123.000
```

注意一个 tricky 的特性：

```
float(f"{int('36'):.2f}")
```

> 输出 36.0
>
> 分析：`int('36')` ： 将字符串转为整数 36
>
> `f"{int('36'):.2f}"` ： 格式化字符串，获得字符串 `'36.00'`
>
> `float(f"{int('36'):.2f}")` ： 将字符串 `'36.00'` 转浮点，结果是 `36.0` ， 因为 Python 的浮点数输出通常只显示必要的位数（有 1 位足以说明它是浮点，再加 0 没有意义）.

所以不用强求用 `float()` ，得到表示两位小数的字符串后直接 `print()` 即可.

```
print(f"{int('36'):.2f}")
```

> 输出 36.00，因为这里直接 `print()` .

③ 处理输入错误

> ② 中提到，`int()` ， `float()` 两者都不能处理无法转为数字的字符串，如果传入无效字符串，会抛出 `ValueError` .

可以使用 `try-except` 处理异常

```
while 1:
    try:
        age = int(input("请输入你的年龄: "))
        print(f"你已经 {age} 岁了。")
        break
    except ValueError:
        print("请输入一个有效的数字。")
```

> 这里 `while` 的作用是，如果用户输入异常数据，让他重新输入，直到输入正常为止.

*python 的 `try-except` 语法

```
try:
    # 可能抛出异常的代码
except SomeException as e:
    # 处理异常
else:
    # 如果没有抛出异常，执行这部分代码 （可选）
finally:
    # 无论是否抛出异常，都会执行这部分代码 （可选）
```

4.2 `eval()`

将传入的字符串当作**有效的 Python 表达式**来求值，并返回结果.

4.2.1 语法

```
eval(expression)
```

`expression`：一个字符串，表示一个有效的表达式.

```
>>> eval('36')
36
>>> eval('2+3*6')
20
```

4.2.2 示例

> 以下几个例子都可以简单地记作：去除引号并执行.

① 计算用户输入的表达式

```
expression = input("请输入一个数学表达式: ")
result = eval(expression)
print(f"结果是: {result}")
```

② 列表、字典等数据结构的动态生成

```
# 将字符串形式的列表转换为实际的列表
my_list = eval("[1, 2, 3, 4]")
print(my_list)  # 输出: [1, 2, 3, 4]
```

③ 动态执行代码

> 注意安全

```
eval("__import__('os').system('rm -rf /')")
```

> 这是一个删除系统文件的代码. 提醒我们不要 `eval()` 不受信任的输入.
>
> 不要运行！不要运行！不要运行！

4.3 File

> File operations

`Python 2` 中，`file` 是内置类型，表示文件对象. 可通过 `type()` 函数看到文件对象的类型是 `file` .

```
# Python 2 示例
f = open('example.txt', 'r')
print(type(f))  # 输出: <type 'file'>
```

`Python 3` 中，`file` 类型被移除了，文件对象的类型变成 `_io.TextIOWrapper` 或 `_io.BufferedReader` ，取决于如何打开文件.

```
# Python 3 示例
f = open('example.txt', 'r')
print(type(f))  # 输出: <class '_io.TextIOWrapper'>
```

4.3.1 打开

语法

要求掌握：

```
file_object = open(filename, mode)
```

完整版：

```
file_object = open(filename, mode='r', buffering=-1, encoding=None, errors=None,
newline=None, closefd=True, opener=None)
```

`filename`：文件路径. 可以是相对路径或绝对路径.

`mode`：指定文件的打开模式，默认`'r'`（只读）.

`encoding`：指定文件的编码格式（如`'utf-8'`），默认是系统的默认编码.

（其他参数待补充.）

mode	含义
`'r'`	只读. 文件必须存在.
`'w'`	写入. 文件存在则清空，不存在则创建.
`'a'`	追加. 指针放在文件末尾，不存在则创建.
`'r+'`	读写. 文件必须存在.
`'rb'`	读取二进制文件.

示例

```
# 打开一个文件进行读取
file = open("example.txt", "r")

# 打开一个文件进行写入（如果文件不存在，将创建它）
file = open("example.txt", "w")

# 打开一个文件以追加模式进行写入
file = open("example.txt", "a")

# 打开一个二进制文件进行读取
file = open("example.bin", "rb")
```

4.3.2 读写

操作	描述
`f = open("filename")`	打开文件，返回文件对象
`f = open("filename", encoding)`	打开文件并指定编码
`f.read()`	读取整个文件内容，返回字符串
`f.read(n)`	读取不超过 `n` 个字符
`f.readline()`	读取下一行内容
`f.readlines()`	读取文件的所有内容并返回一个列表
`f.write(s)`	将字符串 `s` 写入文件
`f.writelines(lst)`	将列表 `lst` 中的每一项写入文件
`f.close()`	关闭文件

> `f.writelines(lst)` 不会分行. 若需要分行，确保每个元素末尾包含换行符.
>
> `f.readline()` 如果某行末尾有换行符（不是最后一行一般都有），会把换行符一起读进去. 若文件为空，返回空字符串.

读取

对于如下的 `example.txt`：

```
Hello, World!
Python is awesome.
```

① `readline` 读取

```
with open("example.txt", "r") as f:
    content = f.readline()  # 第一行
    print(content)
    content = f.readline()  # 第二行
    print(content)
    content = f.readline()  # 没有第三行
    print(content)
```

输出如下：

```
Hello, World!

Python is awesome.

```

> 第三次使用 `.readline()`，指针已在末尾，没有读取任何信息，但是 `print()` 本身输出一个换行符.

② for 循环读取

运行以下内容：

```
>>> f = open('example.txt')
>>> for line in f:
...     print (line)
...
...
Hello, World!

Python is awesome.
>>> for line in f:
...     print (line)
...
...
# nothing happened
```

> `print()` 函数本身会在输出末尾添加一个换行符，而 `.txt` 的第一行末尾也有换行符，因此出现空行.
>
> 注意，第一次迭代时，文件指针从头开始，逐行读取，直到末尾；第二次迭代，指针一开始就在末尾，没有内容可以读取. 因此这个只有第一次使用有效.

以下方法可以避免多余空行：

```
with open("example.txt") as f:
    for line in f:
        print(line.strip())  # 去掉每行的换行符
```

> 关于 `.strip()` 函数，见 `1.2 Strings 字符串` 或 `2.1.5 String Functions`.

输出如下：

```
Hello, World!
Python is awesome.
```

写入

① `.write(string)`

> 将字符串写入文件.

② `.writelines(list_of_strings)`

> 将字符串列表写入文件（不会自动添加换行符）.

```
# 打开文件进行写入
file = open("example.txt", "w")

# 写入单行内容
file.write("Hello, World!\n")

# 写入多行内容
lines = ["First line\n", "Second line\n", "Third line\n"]
file.writelines(lines)

# 关闭文件
file.close()
```

运行后的 `example.txt` ：

```
Hello, World!
First line
Second line
Third line
```

> 注意，是写入模式，不是追加模式，原先内容被清空.
>
> 注意第四行的末尾也有换行符，因此总共五行.

4.3.3 关闭

语法

```
file = open("example.txt", "r")
# 执行一些操作
file.close()
```

4.3.4 try-except 处理异常

> File I/O operations can generate exceptions, an IOError. Exception handling can prevent these errors.

```
try:
    f = open('input.txt')
except IOError as e:
    print ('unable to open the file with error: "{}"'.format(e.args[-1]))
else:
    print ('continue with processing')
    f.close()
print ('continue')
```

如果文件 `'input.txt'` 不存在，输出如下：

```
unable to open the file with error: "No such file or directory"
continue
```

如果文件 `'input.txt'` 存在，输出如下：

```
continue with processing
continue
```

这样保证了无论文件是否正常打开，程序都不会异常终止.

4.3.5 with ... as ... 语句

> A better way is using `with` . Ensure file is closed when the block inside with is exited. But exception handling is still required as needed.

Python 提供更加安全简洁的方式打开文件：使用 `with` 语句（上下文管理器）. 它会自动管理文件的打开和关闭，即使在文件操作过程中抛出异常，也能保证文件被正确关闭.

> 但是异常仍然需要处理.

```
with open('example.txt', 'w', encoding='utf-8') as outf:
    # perform file operation
    outf.write("Hello world")

# 文件在 with 块执行完毕后会自动关闭

# Check if the file has been automatically closed.
print(outf.closed) # prints True
print ('continue with processing')
```

`outf`：文件对象.

输出如下：

```
True
continue with processing
```

运行后，`example.txt` 如下：

```
Hello world
```

4.3.6 OS command

> Useful OS command can be executed from python by including os module.

Python 通过 `import os`，可以执行系统指令.

```
>>> import os
>>> os.remove("gone.txt") # removing file
>>> os.curdir # get current directory
'.'
>>> os.rename('oldfile.txt', 'newfile.txt')
>>>
```

> `os.remove`：只能移除存在的文件，否则报错.
>
> `os.curdir`：输出单个点字符串，表示当前目录.
>
> `os.rename`：只能重命名存在的文件，否则报错.

4.3.7 Standard I/O

标准输出 & 标准输入

`print()` writes characters to a file normally attached to display window.

`Input()` functions read from a file attached to keyboard.

These files can be accessed through `sys` module.

Input file : `sys.stdin`, output file: `sys.stdout`, error messages: `sys.stderr`

`stderr` normally goes also to `stdout`

> 待补充.

Can change these settings through `sys`.

> 要调用标准输入输出，不要忘记 `import sys` .

```
import sys
sys.stdout = open('output.txt', 'w')
sys.stderr = open('error.txt', 'w')
print ("see where this goes")
print (5/4)
print (7.0/0)
sys.stdout.close()
sys.stderr.close()
```

> 这是课件给的代码，实际上它只能生成两个空文件，原因如下：
>
> 当我们将输出重定向到文件地址时，Python 会将数据先写入缓冲区，而不是立即写入文件. 执行 `.flush()` 和 `.close()` 时才会刷新缓冲区，把缓冲区的内容写入文件.
>
> 看这段代码，前两个 `print` 成功写入缓冲区，但是第三个 `print` 抛出了 `ZeroDivisionError` ，却没有捕获，异常写入 `error.txt` 的缓冲区. 但此时整个程序异常终止，两个文件的缓冲区都来不及刷新，所以运行结束后只有两个空文件.

```
import sys
sys.stdout = open('output.txt', 'w')
sys.stderr = open('error.txt', 'w')
print ("see where this goes")
print (5/4)
print (7.0/0)
sys.stdout.flush()
sys.stderr.flush()
sys.stdout.close()
sys.stderr.close()
```

> 这样也是错的，因为 `.flush()` 来不及执行，程序就终止了，缓冲区仍然没有刷新.

正确代码如下：

```
import sys

# 重定向标准输出和标准错误到文件
sys.stdout = open('output.txt', 'w')
sys.stderr = open('error.txt', 'w')

# 输出内容
print("see where this goes")
print(5 / 4)

try:
    print(7.0 / 0)  # 这里会抛出 ZeroDivisionError
except ZeroDivisionError as e:
    print(f"Error: {e}", file=sys.stderr)

# 手动刷新并关闭文件
sys.stdout.flush()
sys.stderr.flush()
sys.stdout.close()
sys.stderr.close()
```

更好的方法是使用 `with` 语句：

```
import sys

# 使用 with 语句管理文件
with open('output.txt', 'w') as out, open('error.txt', 'w') as err:
    sys.stdout = out
    sys.stderr = err

    print("see where this goes")
    print(5 / 4)

    try:
        print(7.0 / 0)  # 这里会抛出 ZeroDivisionError
    except ZeroDivisionError as e:
        print(f"Error: {e}", file=sys.stderr)

# 不需要手动 close，with 会自动关闭并刷新
```

指定输出路径后，控制台不再显示内容. 程序运行后，同目录下有两个文件：

`output.txt`

```
see where this goes
1.25
```

`error.txt`

```
Error: float division by zero
```

> 注意 `print` 自带换行符，两个文件都有一个空行.

4.3.8 OS Functions

`sys` 中也包含一些系统级函数.

① `sys.exit()`

```
import sys

print("before exit")
sys.exit("exit now")
print("after exit") # 程序已终止，无法执行
```

输出如下：

```
before exit
SystemExit: exit now
```

② `sys.argv`

> 不接受参数

`sys.argv` 是一个列表，存储命令行传递给脚本的参数（包括脚本文件本身）. 它允许用户通过命令行给脚本传递参数，并在脚本中访问这些参数.

```
import sys

print ('argument of program are ', sys.argv)
```

输出如下：

```
argument of program are  ['E:/材料/大学/大二 term 1/CSCI 2040
Python/Lectures/argv_test.py']
```

> 输出绝对路径，因为我不是在脚本所在文件夹下运行，而是从其他位置调用脚本并运行.

如果直接在同文件夹下运行，输出如下：

```
argument of program are  ['argv_test.py']
```

4.4 Pickle

`pickle` 是 Python 模块，用于 **序列化（serialization）** 和 **反序列化（deserialization）** Python 对象. 它可以将 Python 的复杂对象（列表、字典、自定义对象等）转换为字节流，便于存储到文件或通过网络传输；也可将字节流重新加载为原先的 Python 对象.

> 注意：需要 `import pickle` .
>
> 字节流，byte stream，二进制形式表示，通常写入二进制文件. 如果将字节流写入文本文件，Python 会尝试将字节数据解码为文本（如 `UTF-8` 编码），可能导致：
>
> 数据被错误解码，或
>
> 数据的原始格式被破坏（某些字节解码时会被替换为 `?` 或其他乱码）.

4.4.1 示例

运行该程序：

```
import pickle # import to use
listOne = list() # we can use list() to create a list
listTwo = list()
listOne.append( 12 )
listTwo.append( 'abc' )
listOne.append( 23 )
listOne.pop()
f = open('pickle1.pyp','wb') # store the variables in a file
pickle.dump([listOne, listTwo], f)
```

然后可以执行下列指令并返回相应结果：

```
>>> import pickle
>>> f = open('pickle1.pyp', "rb")
>>> [listOne, listTwo] = pickle.load(f)
>>> print (listOne.pop())
12
>>> print (listTwo.pop())
abc
```

4.4.2 `Pickle` 的函数

> 待补充.

考试考察上面出现的两个，即

`pickle.dump(obj, file)`：将对象 `obj` 序列化并写入文件对象 `file`.

> 注意写入文件对象前，必须先用 `file = open('filename.pyp','wb')` 给 `file` 赋一个 `'wb'` 属性的文件对象.

`pickle.load(file)`：从文件对象 `file` 中反序列化并返回对象.

> 同理，读取文件对象前，也要先给 `file` 赋一个 `'rb'` 属性的文件对象.

4.5 Dictionary

Indexed data structure - uses also square bracket notation.

> key-value pair，键值对，键是索引，值与键关联.

Any immutable type can be used as index.

> 索引（键）必须是不可变对象（列表不能作为键）.
>
> 常用字符串作为键.

Braces create dictionary.

> 大括号创建字典.

4.5.1 示例

```
dct = { } # create new dictionary
dct['name'] = "Donald Duck"
dct['age'] = 90
dct['eyes'] = "black"

print (dct['name'])
print (dct.get('age'))
print (dct['weight'])
```

> `.get`：如果键不存在，不会抛出错误，而是返回指定值（没指定则默认 None）.
>
> 注意要先创建词典本身.

输出如下：

```
Donald Duck
90
Traceback (most recent call last):
  File "E:/材料/大学/大二 term 1/CSCI 2040 Python/Lectures/Dictionary_test.py",
line 8, in <module>
    print (dct['weight'])
KeyError: 'weight'
```

直接初始化

```
>>> info = {'name':'Batman', 'age':82, 'weight':180}
>>> print (info['name'])
Batman
```

键不能重复

> No duplicate keys allowed
>
> 程序不会出错，但是新的会覆盖掉旧的.
>
> 如果你的目的就是更新键的值，反而可以利用这个特性（见 `4.5.3 修改键的值`）.

键是不可变对象

> Keys must be immutable i.e. lists not allowed

```
>>> dict = {['Name']: 'Zara', 'Age': 7}
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    dict = {['Name']: 'Zara', 'Age': 7}
TypeError: unhashable type: 'list'
>>>
```

4.5.2 get method

Exception when no value with designated key.

Can be prevented by using built-in `get` method to check.

```
dct = { } # create new dictionary
dct['name'] = "Donald Duck"
dct['age'] = 90
dct['eyes'] = "black"

print (dct['name'])
print (dct.get('age'))
print (dct.get('weight', 0))
```

> 0 is default value

输出如下：

```
Donald Duck
90
0
```

4.5.3 修改键的值

```
dct = { } # create new dictionary
dct['name'] = "Donald Duck"
dct['age'] = 90
dct['eyes'] = "black"

dct['age'] = 18 # 可修改值
print(dct['age'])
```

输出如下：

```
18
```

4.5.4 `del` 删除 key

键是不可变对象，但可以用一些方法**间接修改**.

`del` 关键字用于 delete an element from list

> 删除键的同时，值也被删除.

运行代码：

```
dct = { } # create new dictionary
dct['name'] = "Donald Duck"
dct['age'] = 90
dct['eyes'] = "black"

del dct['age']
print (dct['age'])
```

输出如下:

```
Traceback (most recent call last):
  File "E:/材料/大学/大二 term 1/CSCI 2040 Python/Lectures/delete_key_test.py",
line 7, in <module>
    print (dct['age'])
KeyError: 'age'
```

4.5.5 Dictionary Operations

Operation	Description
`len(d)`	Number of elements in `d`.
`d[k]`	Item in `d` with key `k`. If `k` is not found in `d`, raises a `KeyError`.
`d[k] = v`	Set item in `d` with key `k` to `v`.
`d.clear()`	Remove all items from dictionary `d`.
`d.copy()`	Make a shallow copy of `d`.
`k in d`	Return `True` if `d` has key `k`, `False` otherwise.
`d.items()`	Return a list of `(key, value)` pairs.
`d.keys()`	Return a list of keys in `d`.
`d.values()`	Return a list of values in `d`.
`d.get(k)`	Same as `d[k]` except if `k` is not found in `d`, returns `None`.
`d.get(k, v)`	Return `d[k]` if `k` is valid, otherwise return `v`.

Notes:

- `d.items()` returns a **list** in Python 2.7 or earlier, but returns a **view** in Python 3.x or later.
- `d.keys()` and `d.values()` behave similarly:

- They return a **list** in Python 2.7 or earlier.
- They return a **view** in Python 3.x or later.

4.5.6 视图对象

`4.5.5` 中提到，Dict methods return "views" instead of lists in 3.X. views like a window on the keys and values (or items) of a dictionary.

在 Python 3 中，字典的 `keys()`、`values()` 和 `items()` 方法返回的不是列表（像 Python 2 中那样），而是 **视图对象（views）**. 这是一种动态、高效的方式，用来访问字典的键、值或键值对.

① **视图（view）** 是一种动态的视图对象，相当于一个窗口，可以实时反映字典当前的状态.

② **动态性**：视图会随字典的变化而更新，例如新增或删除键值对后，视图内容会立即反映最新的字典状态.

③ **不是列表**：视图本身不是列表，但它是 **可迭代的**（iterable），你可以用 `for` 循环遍历它.

④ 如果需要将视图转换为列表，可以用 `list()` 显式地转换.

示例

① `dict.keys()`

```
d = {"a": 1, "b": 2, "c": 3}
keys_view = d.keys()
print(keys_view)  # 输出: dict_keys(['a', 'b', 'c'])
```

② `dict.values()`

```
d = {"a": 1, "b": 2, "c": 3}
values_view = d.values()
print(values_view)  # 输出: dict_values([1, 2, 3])
```

③ `dict.items()`

```
d = {"a": 1, "b": 2, "c": 3}
items_view = d.items()
print(items_view)  # 输出: dict_items([('a', 1), ('b', 2), ('c', 3)])
```

④ 动态性展示

```
d = {"a": 1, "b": 2, "c": 3}
keys_view = d.keys()

print(keys_view)  # 输出: dict_keys(['a', 'b', 'c'])

# 修改字典
d["d"] = 4
print(keys_view)  # 输出: dict_keys(['a', 'b', 'c', 'd'])

# 再次修改字典
del d["a"]
print(keys_view)  # 输出: dict_keys(['b', 'c', 'd'])
```

> 动态反映在，如果修改了字典，视图会自动更新，不需要重新创建或手动更新视图.
>
> 注意视图对象的结构，例如 `dict_keys(['b', 'c', 'd'])`

4.5.7 结合列表推导式

> List Comprehension & Dictionary

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

> 见 `3.5 列表推导式` .

Operations on dictionaries performed by selecting values from range of keys, then returning items with selected keys.

> 返回符合条件的键对应的值的列表

```
>>> d = {1:'fred', 7:'sam', 8:'alice', 22:'helen'}
>>> [d[i] for i in d.keys() if i%2==0]
['alice', 'helen']
```

4.5.8 字典推导式

> 待补充.
>
> lab 4 q2 中有用到.

Lecture 5: More on Functions

Lecture 6: Object-Oriented Programming

6.1 OOP 概念

> Java 已学过，这里给一段 Python 的 OOP 代码.

```
# Rocket is a class which simulates a rocket ship
class Rocket():
    def __init__(self): # Each rocket has an (x,y) position, and init to 0
        self.x = 0
        self.y = 0
    def move_up(self): # a method which moves the rocket ship up by 1 unit
        self.y = self.y + 1
# One can *instantiate* an instance of the Rocket class
my_rocket = Rocket() # instantiate an instance
# my_rocket has a copy of each of the class's variables,
# and it can do any action that is defined for the class.
print(my_rocket) # shows that my_rocket is stored at a particular location
print('my_rocket x is = ', my_rocket.x, ", my rocket y is = ", my_rocket.y)
```

输出如下：

```
<__main__.Rocket object at 0x0000027AC16D6AB0>
my_rocket x is =  0 , my rocket y is =  0
```

6.2 OOP 术语

Class

> 类，用 `class` 关键字定义.

Attribute

> 类似 Java 的字段，field
>
> Python 的属性更加动态，实例属性直接在构造函数或其他方法中动态创建，而不用在类中显式定义.
>
> 如果不定义类构造函数（即 `__init__(self, attr1, attr2, ...)` 方法），通过该类创建的实例一开始没有任何属性. 但是之后可以显式地给实例动态添加属性.

Behavior/Method

> 类似 Java 的方法，但 Python 有类方法、实例方法和静态方法，Java 只有类方法（静态方法）和实例方法；Python 的实例方法对应 Java 的实例方法，Python 的类方法对应 Java 的类方法（静态方法），但是 Python 有额外的静态方法（Java中没有对应概念），这是一种无法访问任何属性的方法，无论是类属性还是实例属性.
>
> 此外，Python 还有一些特殊方法（魔术方法），可以重载运算符或让对象具有特定行为，而 Java 不支持运算符重载.
>
> 注意，Python 的类方法或静态方法需要在 `def` 上方添加装饰器.
>
> 类方法：`@classmethod`
>
> 静态方法：`@staticmethod`
>
> 不加装饰器默认为实例方法.

Object

> 类似 Java 的对象 / 实例

6.3 类与对象

> Class, Object / Instance

6.3.1 创建多个实例

创建类的多个实例

```
class Rocket():
    def __init__(self):
        self.x = 0
        self.y = 0
    def move_up(self):
        self.y = self.y + 1

# The following code shows that one can create **multiple instances** of
# Create a fleet of 5 rockets, and store them in a list.
my_rockets = [ ]
for x in range(0,5):
    new_rocket = Rocket()
    my_rockets.append(new_rocket) # my_rocket contains 5 Rocket instances
# Show that each rocket is a separate object.
for rocket in my_rockets:
    print(rocket)
```

输出如下：

```
<__main__.Rocket object at 0x000001DD17CB6B40>
<__main__.Rocket object at 0x000001DD17CB6B70>
<__main__.Rocket object at 0x000001DD17CB6AE0>
<__main__.Rocket object at 0x000001DD17CB6BA0>
<__main__.Rocket object at 0x000001DD17CB6BD0>
```

> 未定义 `__str__` 或 `__repr__` 时，`print(实例)` 会返回 `<类名 object at 内存地址>`.

使用列表推导式创建多个实例，更优雅：

```
class Rocket():
    def __init__(self):
        self.x = 0
        self.y = 0
    def move_up(self):
        self.y = self.y + 1

# The following code shows a more elegant way to create multiple instances using
list comprehension
```

```
my_rockets = [Rocket() for x in range(0,5)]
my_rockets[0].move_up()
my_rockets[1].move_up()
my_rockets[1].move_up()
my_rockets[3].move_up()
my_rockets[3].move_up()
my_rockets[3].move_up()
my_rockets[4].move_up()
my_rockets[4].move_up()
for rocket in my_rockets:
    print('For rocket in memory:', rocket, ', its altitude is: ', rocket.y)
```

输出如下：

```
For rocket in memory: <__main__.Rocket object at 0x000001D298AB6B40> , its
altitude is:  1
For rocket in memory: <__main__.Rocket object at 0x000001D298AB6B70> , its
altitude is:  2
For rocket in memory: <__main__.Rocket object at 0x000001D298AB6BA0> , its
altitude is:  0
For rocket in memory: <__main__.Rocket object at 0x000001D298AB6C00> , its
altitude is:  3
For rocket in memory: <__main__.Rocket object at 0x000001D298AB6C30> , its
altitude is:  2
```

6.3.2 类构造函数 `__init__()`

> 注意，Python 不允许定义多个构造函数，如果试图定义两个构造函数，后定义的会覆盖前面的. 比如你先定义一个传参的构造函数，再定义一个不传参的构造函数，接下来你传参创建一个实例就会报错，因为构造函数被覆盖了.

```
class Rocket():
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def move_up(self):
        self.y += 1

rockets = [ ]
rockets.append(Rocket())
rockets.append(Rocket(0,10))
rockets.append(Rocket(100,0))

for index, rocket in enumerate(rockets):
    print("Rocket {} is at ({}, {})." .format (index, rocket.x, rocket.y))
```

`def __init__(self, x=0, y=0):`

> using keywords with default values
>
> 如果不输入参数，`x` 和 `y` 使用默认值 `0` .
>
> 如果输入一个参数，按照位置分配给 `x` ， `y` 使用默认值 `0` .
>
> 如果输入两个参数，按照位置分配给 `x` 和 `y` .

输出如下：

```
Rocket 0 is at (0, 0).
Rocket 1 is at (0, 10).
Rocket 2 is at (100, 0).
```

6.3.3 对象作为参数

```
from math import sqrt

class Rocket():
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def move_up(self):
        self.y += 1

    def get_distance(self, other_rocket): # other_rocket 是一个对象参数
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

# Make two rockets, at different places.
rocket_0 = Rocket()
rocket_1 = Rocket(10,5)
# Show the distance between them.
distance = rocket_0.get_distance(rocket_0)
print("The rockets are %f units apart." % distance)
distance = rocket_0.get_distance(rocket_1)
print("The rockets are %f units apart." % distance)
```

> `%f` 默认保留六位小数.
>
> % distance 是旧版格式化字符串，现在一般这么写：
>
> `print("The rockets are {:.6f} units apart.".format(distance))`
>
> 或者
>
> `print(f"The rockets are {distance:.6f} units apart.")`

输出如下：

```
The rockets are 0.000000 units apart.
The rockets are 11.180340 units apart.
```

6.3.4 类变量 / 类属性

> Class variables，class attributes, 类似 Java 的 class fields

只在类定义时声明一次，所有实例共用.

```
class Rocket:
    count = 0
    def __init__(self):
        Rocket.count = Rocket.count + 1
```

定义后执行以下代码：

```
>>> a = Rocket()
>>> b = Rocket()
>>> print (Rocket.count) # 可以通过类名访问
2
```

6.3.5 继承

> Inheritance
>
> the new class inherits all of the attributes and behavior of old class
>
> 子类可以重写、拓展、添加方法 / 构造函数，不会影响到父类.

在 Python 3 中：

```
class classname():
class classname:
class classname(object):
```

这三种形式是等价的，它们都定义了一个类，并默认继承自 `object` 类.

如果我们要继承自己或其他开发者编写的类呢?

语法

```
class classname(parent):
```

> `parent` 是你想继承的父类名.

① 继承自己的类

> 要结合 `import` 知识.

准备好 `rocket.py` ：

```
class Rocket():
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def move_up(self):
        self.y += 1
```

然后编写另一个 `.py` 文件（同目录）：

```
from rocket import Rocket

class Shuttle(Rocket):
    def __init__(self, x=0, y=0, flights_completed=0):
        super().__init__(x, y)
        self.flights_completed = flights_completed

shuttle = Shuttle(10, 0, 3)
print(shuttle)
print("This shuttle has x = ", shuttle.x, "y = ", shuttle.y, "# of completed
flights ",
shuttle.flights_completed)
```

> 重写了构造函数.
>
> `super().__init__(x, y)` ：调用父类的构造函数对前两个参数初始化. 注意不需要传 `self` 参数，`super()` 会自动传. 用父类名显式调用才需要传递 `self` .

输出如下：

```
<__main__.Shuttle object at 0x0000024EB6166AE0>
This shuttle has x =  10 y =  0 # of completed flights  3
```

② 继承其他开发者的开源类

> 待补充.

覆写

> Overriding

有时子类需要修改或替换从父类继承的行为，此时子类可以使用相同的名称和参数重新定义函数. 若要调用原始父类函数，必须显式提供类名（或者在大多数情况下使用`super()`）

```
class BankAccount(object):
    def __init__(self, balance=0):
        self.balance = balance
    def deposit (self, amount):
        self.balance = self.balance + amount
    def withdraw (self, amount):
        self.balance = self.balance - amount
    def getBalance(self):
        return self.balance

class CheckAccount(BankAccount):
    def withdraw(self, amount):
        print ('withdrawing ', amount)
        BankAccount.withdraw(self, amount)
```

> 新的`withdraw`方法多了打印功能.
>
> 如果使用`super`，应该是`super().withdraw(amount)`. 不需要传`self`参数，因为`super()`会自动将当前实例`self`作为第一个参数传递给父类，如果再写一个就传了两个`self`，是不对的.

6.3.6 示例：银行账户

> 老生常谈.

```
class BankAccount(object):
    def __init__(self, balance=0):
        self.balance = balance
    def deposit (self, amount):
        self.balance = self.balance + amount
    def withdraw (self, amount):
        self.balance = self.balance - amount
    def getBalance(self):
        return self.balance
my_account1 = BankAccount (200)
print ('my_account 1 balance: ', my_account1.getBalance())
my_account2 = BankAccount ()
print ('my_account 2 balance: ', my_account2.getBalance())
```

> 不加装饰器默认为实例方法.

输出如下：

```
my_account 1 balance:  200
my_account 2 balance:  0
```

6.3.7 传递对象

> 把一个对象赋给一个变量，实际上是传递这个对象的引用. 因此，修改对象的属性会反映在两个变量上，因为它们引用的是同一个对象.

```
class BankAccount(object):
    def __init__(self, balance=0):
        self.balance = balance
    def deposit (self, amount):
        self.balance = self.balance + amount
    def withdraw (self, amount):
        self.balance = self.balance - amount
    def getBalance(self):
        return self.balance

husband_account = BankAccount(500)
wife_account = husband_account
wife_account.withdraw(300)
print("hushand account's balance = ", husband_account.balance)
print("wife account's balance = ", wife_account.balance)
```

输出如下:

```
hushand account's balance =  200
wife account's balance =  200
```

```
class BankAccount(object):
    def __init__(self, balance=0):
        self.balance = balance
    def deposit (self, amount):
        self.balance = self.balance + amount
    def withdraw (self, amount):
        self.balance = self.balance - amount
    def getBalance(self):
        return self.balance

class CheckAccount(BankAccount):
    def __init__(self, initBal=0):
        BankAccount.__init__(self, initBal)
        self.checkRecord = {} # checkRecord, a new dict attribute
    def processCheck(self, number, toWho, amount):
        self.withdraw(amount)
        self.checkRecord[number] = (toWho, amount)
    def checkInfo(self, number):
        if number in self.checkRecord:
```

```
            return self.checkRecord[number]

ca = CheckAccount (1000)
ca.processCheck(100, 'CUHK', 328.)
ca.processCheck(101, 'HK Electric', 452.)
print('Check 101 has information of: ', ca.checkInfo(101))
print ('The current balance is: ', ca.getBalance())
ca.deposit(100)
print('The current balance is: ', ca.getBalance())
```

输出如下：

```
Check 101 has information of:  ('HK Electric', 452.0)
The current balance is:  220.0
The current balance is:  320.0
```

待补充：6.3.8 应用：计算器

> 待补充.

待补充：More on Scope

> 待补充.

Lecture 7: Functional Programming

Lecture 8: NumPy & SciPy in Python

8.1 NumPy

NumPy（Numerical Python）是 Python 中用于科学计算的基础库之一，支持高效的多维数组操作，同时包含各种数学运算、线性代数、随机数生成等功能. 是许多科学计算和数据分析库（如 SciPy、Pandas、TensorFlow）的核心依赖.

> 注意，需要下载.

安装

```
pip install numpy
```

核心功能

① 多维数组对象 `ndarray`

② 广播机制

③ 矢量化运算

④ 常用数学函数

⑤ 线性代数

⑥ 随机数生成

⑦ 高效文件读写

8.1.1 概念

① `ndarray`

a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.

多维数组对象

轴 Axis

dimensions are called axes.

> 轴，每个维度被称为一个轴.

Numpy 数组的轴从 `0` 开始编号.

维数 Rank

number of axes is rank.

> 维数. 数组的秩指它的维数，即轴的数量.

形状 Shape

数组的**形状**是一个元组，表示沿每个轴的长度.

示例

一维数组

```
arr = np.array([1, 2, 1])  # 一维数组
print(arr.shape)  # 输出: (3,)
```

Axis：1 个

> axis 0

Rank：1

Shape：`(3,)`

> 一维数组只有一个轴（`axis 0`），它的长度等于数组中元素的个数.
>
> 一维数组的形状是 `(n,)`，其中 `n` 是该轴的长度.

二维数组

```
arr = np.array([[1.0, 1.0, 2.0], [0.0, 2.0, 1.0]])  # 二维数组
print(arr.shape)  # 输出: (2, 3)
```

Rank：2

Axis：2 个

> axis 0：第一维，行方向，长度为 2.
>
> axis 1：第二维，列方向，长度为 3.

Shape：`(2, 3)`

三维数组

```
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])  # 三维数组
print(arr.shape)  # 输出: (2, 2, 2)
```

Rank：3

Shape：`(2, 2, 2)`

> axis 0：长度为 2（最外层有 2 个数组）
>
> axis 1：长度为 2（每个子数组有 2 行）
>
> axis 2：长度为 2（每行有两个元素）

②③④⑤⑥⑦⑧

8.1.2 基本用法

① 创建数组

`np.array()`

```
import numpy as np

# 创建一维数组
arr1 = np.array([1, 2, 3, 4])
print(arr1)  # 输出: [1 2 3 4]

# 创建二维数组
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
# 输出:
# [[1 2 3]
#  [4 5 6]]

# 创建全零数组（浮点）
zeros = np.zeros((2, 3))
print(zeros)
# 输出:
# [[0. 0. 0.]
#  [0. 0. 0.]]

# 创建全一数组（浮点）
ones = np.ones((3, 3))
print(ones)

# 创建单位矩阵（浮点）
eye = np.eye(3)
print(eye)

# 使用范围创建数组（int）
arange = np.arange(0, 10, 2)  # 起始为0，步长为2
print(arange)  # 输出: [0 2 4 6 8]

# 创建线性等距数组（浮点）
```

```
linspace = np.linspace(0, 1, 5)  # 在0和1之间生成5个等距点
print(linspace)  # 输出: [0.   0.25 0.5  0.75 1.  ]
```

输出如下：

```
[1 2 3 4]
[[1 2 3]
 [4 5 6]]
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[0 2 4 6 8]
[0.   0.25 0.5  0.75 1.  ]
```

可以输入第二个参数控制数据类型：

```
import numpy as np
a = np.array([1, 4, 5, 8], float)
print(a)
print (type(a))
```

输出如下：

```
[1. 4. 5. 8.]
<class 'numpy.ndarray'>
```

② 数组运算

③ 数组索引和切片

```
import numpy as np
a = np.array([1, 4, 5, 8], float)

print(a[:2])
print(a[3])
a[3] = 100.0
print(a)
```

输出如下：

```
[1. 4.]
8.0
[ 1. 4. 5. 100.]
```

切片

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# 访问单个元素
print(arr[0, 1])  # 输出: 2

# 切片操作
print(arr[:, 1])  # 输出: [2 5 8]，选中所有行的第2列
print(arr[1:, :2])  # 输出: [[4 5]
                    #         [7 8]]，选中第2行及以后，前两列
```

> 注意，从 0 开始，左闭右开

高维索引

```
a = np.array([[1,2,3], [4,5,6]], float)
print ('a =', a)
a[0,0] = 15.0 # we can re-assign elements in the array
a[1][0] = 12.0
print('a =', a)
```

> 两种索引方式

输出如下：

```
a = [[1. 2. 3.]
 [4. 5. 6.]]
a = [[15.  2.  3.]
 [12.  5.  6.]]
```

④ 数组形状操作

`np.reshape()`

```
import numpy as np

arr = np.arange(6)  # 创建一维数组 [0, 1, 2, 3, 4, 5]
reshaped_arr = arr.reshape(2, 3)  # 改变形状为 2x3 的二维数组
print(reshaped_arr)
# 输出:
# [[0 1 2]
#  [3 4 5]]

# 转置
print(reshaped_arr.T)
# 输出:
# [[0 3]
#  [1 4]
#  [2 5]]
```

⑤ 统计和数学运算

⑥ 随机数生成

Lecture 9: Visualization in Python

Lab 1

期末

Dear CSCI2040 Students,

This is the last teaching week, and we will conduct the final exam on November 21, 2024 (Thu) evening from 6:30 am to 8:15 pm in person. We held the exam at Science Center (SC) L1

We are sure you all are preparing. Details are as follows:

Exam scope: lecture materials from week 1 to NumPy, excluding Visualization or Supplementary/Appendix.

Remarks: common functions such as print(), open(), list.append(), len(), range(), pickle.dump(), map(), filter(), reduce(), np.array(), np.reshape() etc. should be memorized; uncommon functions such as list.pop(), list.index(), string.strip(), string.rfind(), dict.copy(), etc. description will be given if needed.

Question type: code comprehension, coding, problem-solving, short-answer questions, etc.

Rules and regulations:

- NO electronic, communication, or sound-producing devices are allowed in the exam venue, e.g., computers, mobile phones, smartwatches, earphones, etc. Please do NOT bring them in OR keep them in a shut-down state in your bags.
- closed-book, closed-notes
- sparse seating, i.e., always keep vacant seats on both sides of each student
- bring your student ID card for identity checking during the exam

Please pay attention to all sorts of emergent announcements, should any special arrangements arise due to unexpected conditions. Thank you for your attention and cooperation.

Yours,

CSCI2040 Course Teacher

亲爱的CSCI2040学生们，

这是最后一周的教学，我们将在2024年11月21日（星期四）晚上6:30至8:15进行期末考试，考试将以面对面的方式进行。考试地点在科学中心（SC）L1。

我们相信大家都在准备中。详情如下：

考试范围：从第一周到NumPy的讲课内容，不包括可视化或补充/附录。

备注：常用函数如print()、open()、list.append()、len()、range()、pickle.dump()、map()、filter()、reduce()、np.array()、np.reshape()等应记住；不常用的函数如list.pop()、list.index()、string.strip()、string.rfind()、dict.copy()等，如有需要将提供描述。

题型：代码理解、编码、问题解决、简答题等。

规则和规定：

- 禁止携带任何电子、通信或发声设备进入考场，如计算机、手机、智能手表、耳机等。请不要带入或在包内关闭状态。
- 闭卷，禁止带书、笔记
- 稀疏座位，即每位学生两侧始终保持空座位

- 考试时请携带学生证以便身份核查

请注意所有紧急公告，以防由于意外情况而做出特殊安排。感谢您的关注和合作。

你们的，

CSCI2040课程老师

注意点

1.手写代码

比较相等用两个等号，大于等于不要写成数学的那个

语句结尾不用分号，直接换行

不用代码块，python 用缩进表示块

python的 if while for 要加冒号

关于 a++ 和 ++a

区间是左闭右开！

注意，print 如果打印列表，字符元素会带单引号，字符串元素会带单引号或双引号（仅引起歧义时）；如果直接打印，会省略引号.

> 更准确地说，当 `print()` 容器类型（如 `list`、`tuple`、`dict`、`set` 等）时，这些容器会调用它们内部元素的 `repr()` 方法来显示内容.

print(),

> 见 `Introduction`

open(),

list.append(),

> 2.2.4

len(),

> 2.2.1

range(),

> 2.2.2

pickle.dump(),

map(), filter(),

> 见 `3.7.3 应用`

reduce(),

> lab 4 q2

语法

```
from functools import reduce

reduce(function, iterable[, initializer])
```

`function`：一个接受两个参数的函数，用于对可迭代对象的元素进行二元操作（例如加法、乘法等）。

`iterable`：要进行操作的可迭代对象（如列表、元组等）。

`initializer`（可选）：一个可选的初始值。如果提供了 `initializer`，则它将作为第一次调用 `function` 时的第一个参数，并与 `iterable` 的第一个元素进行运算；如果没有提供 `initializer`，则 `iterable` 的第一个元素将作为第一次调用 `function` 时的第一个参数。

工作原理：`reduce()` 函数会从可迭代对象的第一个和第二个元素开始，将它们作为参数传递给 `function`，计算出一个结果。接着，它会将这个结果与第三个元素再次传递给 `function`，以此类推，直到可迭代对象的所有元素都被处理完，最终返回一个单一的累积值。

np.array(),

np.reshape()