- `for root, dir, file in os.walk(path)` 遍历路径下的文件
- `os.path.join(xx, xx, ...)` 合并多个路径，不同系统下通用
- 跳出两层循环

```
for i in range(1,100):
    for j in range(1,100):
       break
    else:
       continue
    break
```

- 关于类
  - 在 `len()` 函数内部，它自动去调用该对象的 `__len__()` 方法
  - 获得一个对象的所有属性和方法，可以使用 `dir()` 函数
  - 配合 `getattr()` 、 `setattr()` 以及 `hasattr()` ，可以直接操作一个对象的状态
  - `@property` 装饰器负责把一个方法变成属性调用
  - `__str__` ，返回自定义字符串，打印时使用
  - `__iter__()` 方法返回一个迭代对象，Python的for循环就会不断调用该迭代对象的 `__next__()` 方法拿到循环的下一个值，直到遇到 `StopIteration` 错误时退出循环
  - 像list那样按照下标取出元素，需要实现 `__getitem__()` 方法，按下标访问任意一项
- `zip()` 实现矩阵的行列互换

`zip()` 函数用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表

```
>>> a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zip(*a)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
>>> map(list,zip(*a))
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

- `eval()` 与 `exec()` 的区别
  - `eval()` 函数只能计算单个表达式的值，而 `exec()` 函数可以动态运行代码段。
  - `eval()` 函数可以有返回值，而 `exec()` 函数返回值永远为 `None` 。
  - 与之相对的， `str()` 和 `repr()`
- `P = {s : {a : [] for a in range(3)} for s in range(4)}` 二维数组的字典实现
- 使用两列数据创建字典

```
keys = ['1001', '1002', '1003']
values = ['骆昊', '王大锤', '白元芳']
d = dict(zip(keys, values))
print(d)
```

- 与命令行的交互，参考

```
import argparse
parse = argparse.ArgumentParser()

parse.add_argument("--bool", help="Whether to pirnt sth")
parse.add_argument("--rate", type=float, default=0.01, help="initial rate")
parse.add_argument("--choice", choices=[0, 1], help="you can only input 0 or 1.")
parse.add_argument("--need", required=True, help="give it a value")

args = parse.parse_args()
if args.bool:
    print("bool: ", args.bool)
if args.rate:
    print("rate: ", args.rate)
if args.choice:
    print("choice: ", args.choice)
if args.need:
```

```
    print("need: ", args.need)
```

一般把可选参数最后add，不可选参数放前面add；在命令行里面输入代码时，程序会先赋值先add的参数

- `pop()` 或 `remove()` 方法删除数组元素，这两个函数之间的区别在于前者返回已删除的值，而后者则不返回
- 多进程需要在main函数中运行
- The `__str__()` and `__repr__()` methods deal with how objects are presented as strings, so you'll need to make sure you include at least one of those methods in your class definition. If you have to pick one, go with `__repr__()` because it can be used in place of `__str__()`.
- list前加 `*`，将列表解开成独立的参数，用于传入函数；dict前加 `**`，传入函数，key作为变量，value作为相应key的值

```
def myfunc(x, y, z):
    print(x, y, z)

tuple_vec = (1, 0, 1)
dict_vec = {'x': 1, 'y': 0, 'z': 1}

>>> myfunc(*tuple_vec)
1, 0, 1

>>> myfunc(**dict_vec)
1, 0, 1
```

- `[P1, P2, P3, ...Pn].sort(key=lambda x: (x.first_key, x.second_key, x.third_key, ...))` 主次关键字排序
- 字典， `setdefault()` 方法设置默认值，或者使用 `collections.defaultdict(type)`
- zip使用多个序列作为参数，然后返回元组的列表，将这些序列中的并排元素一一配对。当zip的多个参数长度不同时，zip会以最短序列的长度为准来截断所得到的元组
- Python dictionaries check for equality and compare the hash value (even if they are of different types, as is the case for 1 and 1.0) to determine if two keys are the same.
- namedtuples can be a great alternative to defining a class manually

```
# Why Python is Great: Namedtuples
# Using namedtuple is way shorter than
# defining a class manually:
>>> from collections import namedtuple
>>> Car = namedtuple('Car', 'color mileage')

# Our new "Car" class works as expected:
>>> my_car = Car('red', 3812.4)
>>> my_car.color
'red'
>>> my_car.mileage
3812.4

# We get a nice string repr for free:
>>> my_car
Car(color='red' , mileage=3812.4)

# Like tuples, namedtuples are immutable:
>>> my_car.color = 'blue'
AttributeError: "can't set attribute"
```

- How to sort a Python dict by value

```
# (== get a representation sorted by value)

>>> xs = {'a': 4, 'b': 3, 'c': 2, 'd': 1}

>>> sorted(xs.items(), key=lambda x: x[1])
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]

# Or:

>>> import operator
>>> sorted(xs.items(), key=operator.itemgetter(1))
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

- merge two dictionaries

```
>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

- A lambda function is a small anonymous function.

  `lambda arguments:expression`

- 一行方法
  - 快速漂亮的从文件打印出 `json` 数据, `cat file.json | python -m json.tool`
  - 脚本性能分析, `python -m cProfile my_script.py`
- `collections.OrderedDict` , `OrderedDict` preserves the order in which the keys are inserted. A regular dict doesn't track the insertion order, and iterating it gives the values in an arbitrary order.
- `collections.namedtuple` , 用来创建一个tuple的子类，其可以通过属性名称访问tuple中的元素，便于理解
- The glob module finds all the pathnames matching a specified pattern.

  支持 `*` , `?` , `[]` 这三个通配符； `glob.glob('*.jpg')`
- Python to process things in parallel

```
import glob
import os
import cv2
import concurrent.futures


def load_and_resize(image_filename):
  ### Read in the image data
  img = cv2.imread(image_filename)

  ### Resize the image
  img = cv2.resize(img, (600, 600))


### Create a pool of processes. By default, one is created for each CPU in your machine.
with concurrent.futures.ProcessPoolExecutor() as executor:
    ### Get a list of files to process
    image_files = glob.glob("*.jpg")

    ### Process the list of files, but split the work across the process pool to use all CPUs
    ### Loop through all jpg files in the current folder
    ### Resize each one to size 600x600
    executor.map(load_and_resize, image_files)
```

- os 不错的用法

```
# Execute a shell command
os.system("echo 'My name is bob the builder'")
```

```
# List all of the files and sub-directories in a particular folder
os.listdir("Documents")
# Create a single folder
os.mkdir("Data Science Projects")
# Create folders recursively
# The below line creates a folder "Documents" with a subfolder
# inside it called "Data Science Projects" with a subfolder inside that one called "Project 1"
os.makedirs("Documents/Data Science Projects/Project 1")
# Delete a file
os.remove("data.txt")
# Delete a folder
os.rmdir("Data Science Projects")
# Delete directories recursively.
os.removedirs("Documents/Data Science Projects/Project 1")
# Rename a file or folder
os.rename("My Documents", "Your Documents")
# Return the current working directory
os.getcwd()
# Get the directory and file name from a full path
file = os.path.basename(full_path)
folder = os.path.dirname(full_path)
# Check if a file or folder exists
os.path.exists(full_path)
# Get the extension of a file
name, extension = os.path.splitext(file)
```

- Advanced unpacking from lists

```
>>> a, *b, c = range(8)
>>> a
0
>>> b
[1, 2, 3, 4, 5, 6]
>>> c
7
```

- lesser known Math functions

```
# Getting both the faction and integer parts
>>> math.modf(96.12)
(0.1200000000, 96.0)
# Computes the Euclidean norm sqrt(x*x + y*y)
>>> math.hypot(2, 3)
3.6056
```

- `bool` 是 `int` 的子类， `isinstance(True, int)` 为真， `type(True) == int` 为假
- Check if two strings are anagrams.

```
from collections import Counter

def anagram(first, second):
    return Counter(first) == Counter(second)
```

- `sys.getsizeof(variable)` memory usage of an object
- multiple functions inside a single line

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

a, b = 4, 5
print((subtract if a > b else add)(a, b)) # 9
```

- merge two dictionaries

```
def merge_dictionaries(a, b)
   return {**a, **b}

a = { 'x': 1, 'y': 2}
b = { 'y': 3, 'z': 4}
print(merge_dictionaries(a, b)) # {'y': 3, 'x': 1, 'z': 4}
```

- Convert two lists into a dictionary

```
def to_dictionary(keys, values):
    return dict(zip(keys, values))

keys = ["a", "b", "c"]
values = [2, 3, 4]
print(to_dictionary(keys, values)) # {'a': 2, 'c': 4, 'b': 3}
```

- the most frequent element that appears in a list

```
def most_frequent(list):
    return max(set(list), key = list.count)

list = [1,2,1,2,3,2,1,4,2]
most_frequent(list)
```

- Get default value for missing keys

```
d = {'a': 1, 'b': 2}

print(d.get('c', 3)) # 3
```

- `is` expressions evaluate to True if two variables point to the same object; `==` evaluates to True if the objects referred to by the variables are equal
- Sanitizing String Input

```
user_input = "This\nstring has\tsome whitespaces...\r\n"

character_map = {
    ord('\n') : ' ',
    ord('\t') : ' ',
    ord('\r') : None
}
user_input.translate(character_map)  # This string has some whitespaces... "
```

- Taking Slice of an Iterator

```
import itertools

s = itertools.islice(range(50), 10, 20)
for val in s:
    print(val)
# 10 11 ... 19
```

- Functions with only Keyword Arguments (kwargs)

It can be helpful to create function that only takes keyword arguments to provide (force)

```
def test(*, a, b):
    pass

test("value for a", "value for b")  # TypeError: test() takes 0 positional arguments...
test(a="value", b="value 2")  # Works...
```

- Creating Object That Supports `with` Statements

```
class Connection:
    def __init__(self):
        ...

    def __enter__(self):
        # Initialize connection...

    def __exit__(self, type, value, traceback):
        # Close connection...

with Connection() as c:
    # __enter__() executes
    ...
    # conn.__exit__() executes
```

```
from contextlib import contextmanager

@contextmanager
def tag(name):
    print(f"<{name}>")
    yield
    print(f"")

with tag("h1"):
    print("This is Title.")
```

The snippet above implements the content management protocol using contextmanager manager decorator. The first part of the tag function (before `yield`) is executed when entering the with block, then the block is executed and finally rest of the tag function is executed.

- Limiting CPU and Memory Usage

```
import signal
import resource
import os

# To Limit CPU time
def time_exceeded(signo, frame):
    print("CPU exceeded...")
    raise SystemExit(1)

def set_max_runtime(seconds):
    # Install the signal handler and set a resource limit
    soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
    resource.setrlimit(resource.RLIMIT_CPU, (seconds, hard))
    signal.signal(signal.SIGXCPU, time_exceeded)

# To limit memory usage
def set_max_memory(size):
    soft, hard = resource.getrlimit(resource.RLIMIT_AS)
    resource.setrlimit(resource.RLIMIT_AS, (size, hard))
```

- Comparison Operators the Easy Way

```
from functools import total_ordering

@total_ordering
class Number:
```

```
    def __init__(self, value):
        self.value = value

    def __lt__(self, other):
        return self.value < other.value

    def __eq__(self, other):
        return self.value == other.value

print(Number(20) > Number(3))
print(Number(1) < Number(5))
print(Number(15) >= Number(15))
print(Number(10) <= Number(2))
```

`total_ordering` decorator is used to simplify the process of implementing ordering of instances for our class. It's only needed to define `__lt__` and `__eq__`, which is the minimum needed for mapping of remaining operations and that's the job of decorator - it fills the gaps for us.

- `@classmethod` vs `@staticmethod` vs "plain" methods

```
class MyClass:
    def method(self):
        """
        Instance methods need a class instance and can access the instance through `self`.
        """
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        """
        Class methods don't need a class instance.
        They can't access the instance (self) but they have access to the class itself via `cls`.
        """
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        """
        Static methods don't have access to `cls` or `self`.
        They work like regular functions but belong to the class's namespace.
        """
        return 'static method called'

# All methods types can be
# called on a class instance:
>>> obj = MyClass()
>>> obj.method()
('instance method called', <MyClass instance at 0x1019381b8>)
>>> obj.classmethod()
('class method called', <class MyClass at 0x101a2f4c8>)
>>> obj.staticmethod()
'static method called'

# Calling instance methods fails
# if we only have the class object:
>>> MyClass.classmethod()
('class method called', <class MyClass at 0x101a2f4c8>)
>>> MyClass.staticmethod()
'static method called'
>>> MyClass.method()
TypeError:
    "unbound method method() must be called with MyClass "
    "instance as first argument (got nothing instead)"
```

- `itertools.permutations()` 排列
- Finding the most common elements in an iterable

```
>>> import collections
>>> c = collections.Counter('helloworld')

>>> c
Counter({'l': 3, 'o': 2, 'e': 1, 'd': 1, 'h': 1, 'r': 1, 'w': 1})

>>> c.most_common(3)
[('l', 3), ('o', 2), ('e', 1)]
```

- `vals = [expression for value in collection if condition]`
- `python -m http.server`
- `from collections import Counter` 计数函数
- 几重循环改写

```
for shape in [True, False]:
    for weight in range(1, 5):
        function(shape, weight)

from itertools import product
function(shape, weight) for weight, shape in product([True, False], range(1, 5))
```

- `map()` 函数接收两个参数，一个是函数，一个是 `Iterable` ，map将传入的函数依次作用到序列的每个元素，并把结果作为新的 `Iterator` 返回
- `reduce()` 把一个函数作用在一个序列 `[x1, x2, x3, ...]` 上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算 `reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)`
- 把 `str` 转换为 `int`

```
>>> from functools import reduce
>>> def fn(x, y):
...     return x * 10 + y
...
>>> def char2num(s):
...     digits = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}
...     return digits[s]
...
>>> reduce(fn, map(char2num, '13579'))
13579
```

- `filter()` 接收一个函数和一个序列，用于过滤序列。 `filter()` 把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素
- `functools.partial()` 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

- 带yield的函数是一个生成器，而不是一个函数了，这个生成器有一个函数就是next函数，next就相当于“下一步”生成哪个数，这一次的next开始的地方是接着上一次的next停止的地方执行的
- `globals()` returns a dict with all global variables in the current scope; `locals()` returns a dict with all local variables in the current scope
- `threading.Thread(target=function, args=(), kwargs={})`
- `python3 -m venv ./venv` , `source ./venv/bin/activate` 虚拟环境
- Python 中的变量更像是是个标签；给变量赋值，就是把标签贴在一个物体上；再次赋值就是把标签贴在另一个物体上。变量不存在实体，它仅仅是一个标签，一旦赋值就被设置到另一个物体上，不变的是那些物体。
- Python 里的参数是通过赋值传递的。函数的返回值 return，也相当于是一次赋值。

Since assignment just creates references to objects, there's no alias between an argument name in the caller and callee, and so no call-by-reference per se. 赋值是创建了一份对象的引用（也就是地址），形参和实参之间不存在别名的关系，本质上不存在引用传递。

- 强制指定参数名

```
# In Python 3 you can use a bare "*" asterisk
# in function parameter lists to force the
# caller to use keyword arguments for certain
# parameters:

>>> def f(a, b, *, c='x', d='y', e='z'):
...     return 'Hello'

# To pass the value for c, d, and e you
# will need to explicitly pass it as
# "key=value" named arguments:
>>> f(1, 2, 'p', 'q', 'v')
TypeError:
"f() takes 2 positional arguments but 5 were given"

>>> f(1, 2, c='p', d='q',e='v')
'Hello'
```

- to be continued