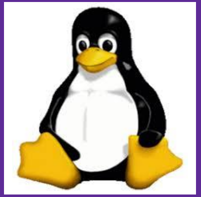
Introduction to Computation

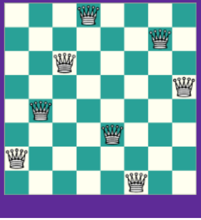
Autumn, 2023

Prof. Fan Cheng

Shanghai Jiao Tong University

chengfan85@gmail.com
https://github.com/ichengfan/itc





8

Outline

- Review of Past Lectures
- Style Guide for Python Code
- Practice problems

Review of Past Lectures

Programming language

- In principle, all the popular PLs have the following elements:
    - Input and output
        - input(), print()
    - Types and variables
        - type(), id(), int(), float(), str(), chr(), ord()
    - Basic expressions: logic, mathematics
    - Conditional expression
        - Code block and indentation
    - Loop expression
        - while, for, break
    - Function
        - Parameters and return values
    - File
        - To be introduced

- Advanced
    - Class and OO
    - Exception
    - Standard library
- When you have become familiar with one PL, you could learn another shortly
    - Don’t learn programming languages but learn how to program
    - You should master several PLs and use them as your primary tools
    - Learn C in 2 two days. ☺

Practice makes perfect

Bug-free code and Debug

- 反复思考程序大的框架，谋定而后动
  - 模块、类、函数
  - 算法、数据结构
- 能正确运行的代码才是好的代码
  - 先实现功能，再优化性能，不要提前优化
  - 提前优化会把系统实现复杂化
- 保证代码结构的清晰和简介
  - 循环深度不超过两轮
  - 单个函数的长度不要过长（25-35行）
  - 多用函数、类、包等机制来隔离代码
  - Zen of Python
- 多试运行
  - 一边编码，一边试运行
  - 3-5行运行一次，看看输出是否正确
  - 注意边际输入：:x=[],“”,(,)
  - 实现每个功能后都做调试，保证前面不错
  - 一个大的功能完成后先反复试运行
- 调试
  - print每个中间的重要数据，保证中间状态正确
  - IDE提供的debug功能，但不建议初学者用这个
  - 初学者的代码可能不超过300行，可以依靠观察能力和print来debug

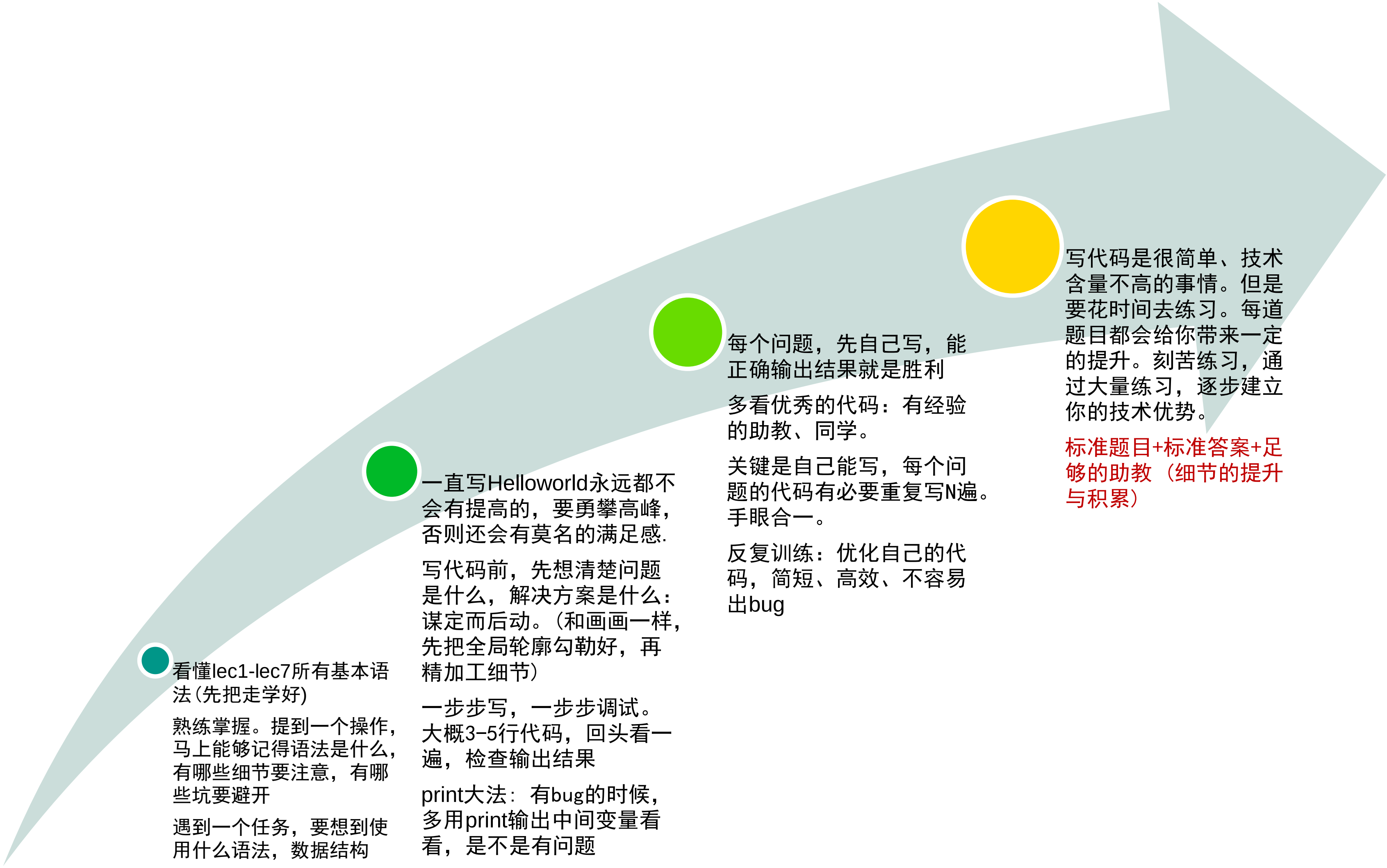
Common bugs

- TypeError
- NameError
- None

Leetcode练习

- 从简单题目开始
  - 自己写代码，能写出来就可以了
  - AC就是最好的
- 困难的题目
  - 看参考样例
  - 看懂、自己能仿照写
  - 回过头自己多写几遍
  - 不会写的原因是写的少了，见得少了，想的少了：maturity
- 可以先在vscode上写好，vscode可以提供一些辅助功能
- 反复练习：提升速度、减少bug
  - 刚开始总是困难的，怎么写怎么错
  - 当你超越50题的时候就会焕然一新
  - 日积月累，1-2个月可以看到效果
  - 我们不是为了期末考试而设置课程
- 目标：100+100+100

不贪多，每天保证3-5道就足够了
每道题务必做透
刚开始速度慢，一天3题，熟练了，可以远远超过3题
2个月的练习到期末前大概可以做完200道



看懂lec1-lec7所有基本语法(先把走学好)

熟练掌握。提到一个操作，马上能够记得语法是什么，有哪些细节要注意，有哪些坑要避开

遇到一个任务，要想到使用什么语法，数据结构

一直写Helloworld永远都不会有提高的，要勇攀高峰，否则还会有莫名的满足感.

写代码前，先想清楚问题是什么，解决方案是什么：谋定而后动。(和画画一样，先把全局轮廓勾勒好，再精加工细节)

一步步写，一步步调试。大概3-5行代码，回头看一遍，检查输出结果

print大法：有bug的时候，多用print输出中间变量看看，是不是有问题

每个问题，先自己写，能正确输出结果就是胜利

多看优秀的代码：有经验的助教、同学。

关键是自己能写，每个问题的代码有必要重复写N遍。手眼合一。

反复训练：优化自己的代码，简短、高效、不容易出bug

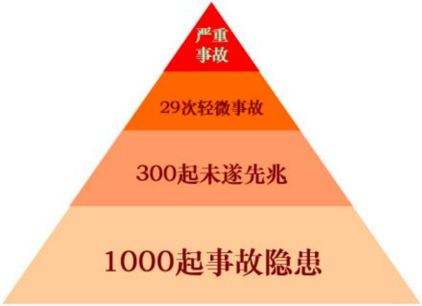
写代码是很简单、技术含量不高的事情。但是要花时间去练习。每道题目都会给你带来一定的提升。刻苦练习，通过大量练习，逐步建立你的技术优势。

标准题目+标准答案+足够的助教（细节的提升与积累）

Style Guide for Python Code

Python规范

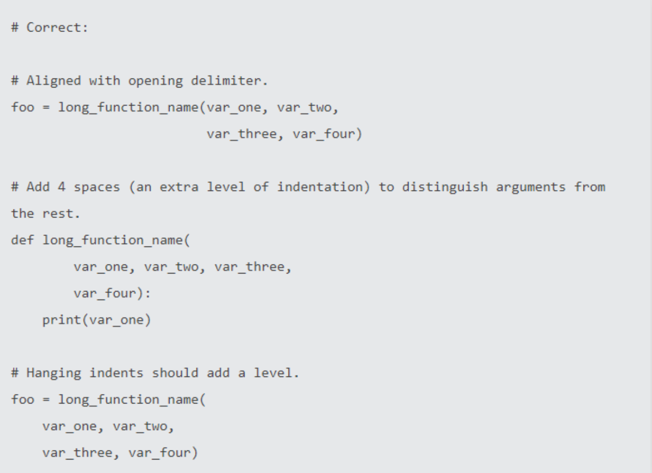
- One of Guido's key insights is that code is read much more often than it is written
- import this: Readability counts
- PEP 8 -- Style Guide for Python Code
  - PEP: Python Enhancement Proposal
  - https://www.python.org/dev/peps/pep-0008/
  - This document gives coding conventions for the Python code comprising the standard library in the main Python distribution
- 除了语法上面合格，还要在风格上面保持一致
- 安全生产：不带电操作；电闸的开关为什么要挂在上面，而不是下面？
- 海恩法则，是航空界关于飞行安全的法则
- 海恩法则指出：每一起严重事故的背后，必然有29次轻微事故和300起未遂先兆以及1000起事故隐患
- 每一个规范，都是血泪教训



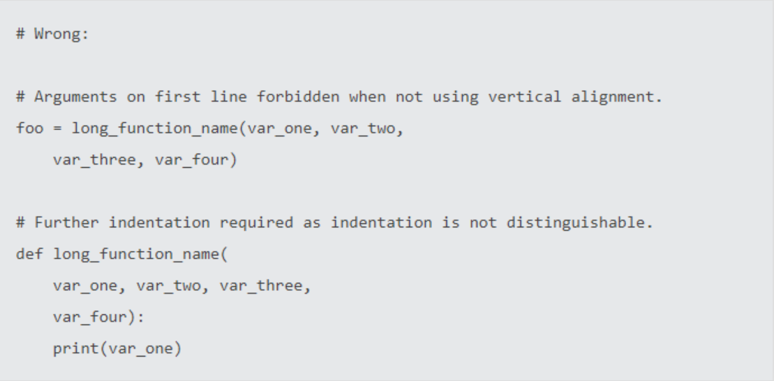
- Introduction
- A Foolish Consistency is the Hobgoblin of Little Minds
- Code Lay-out
    - Indentation
    - Tabs or Spaces?
    - Maximum Line Length
    - Should a Line Break Before or After a Binary Operator?
    - Blank Lines
    - Source File Encoding
    - Imports
    - Module Level Dunder Names
- String Quotes
- Whitespace in Expressions and Statements
    - Pet Peeves
    - Other Recommendations
- When to Use Trailing Commas
- Comments
    - Block Comments
    - Inline Comments
    - Documentation Strings
- Naming Conventions
    - Overriding Principle
    - Descriptive: Naming Styles
    - Prescriptive: Naming Conventions
        - Names to Avoid
        - ASCII Compatibility
        - Package and Module Names
        - Class Names
        - Type Variable Names
        - Exception Names
        - Global Variable Names
        - Function and Variable Names
        - Function and Method Arguments
        - Method Names and Instance Variables
        - Constants
        - Designing for Inheritance
    - Public and Internal Interfaces
- Programming Recommendations
    - Function Annotations
    - Variable Annotations
- References
- Copyright

Indentation

- Use 4 spaces per indentation level
- Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line



```
# Correct:

# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distinguish arguments from
the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

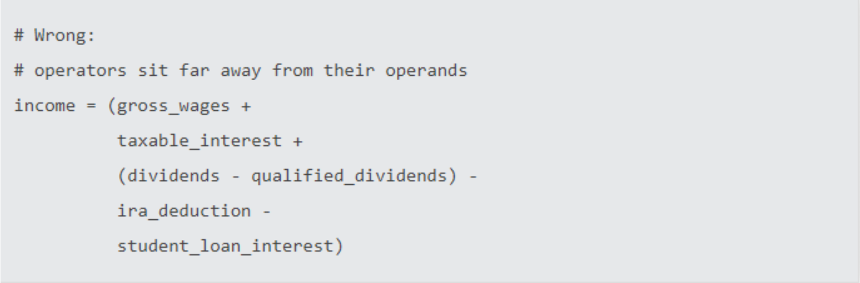


```
# Wrong:

# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

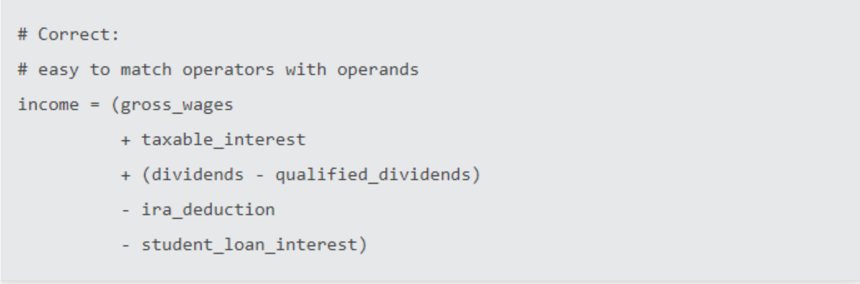
怎么清晰怎么来

Binary Operator

- Donald Knuth explains the traditional rule in his Computers and Typesetting series: "Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations"



```
# Wrong:
# operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

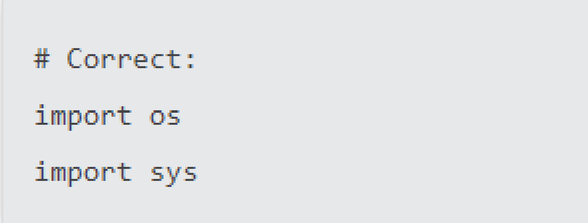


```
# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

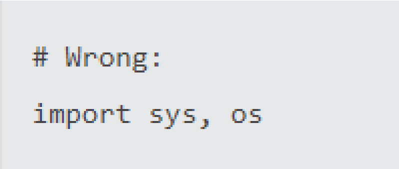
怎么漂亮怎么来

import

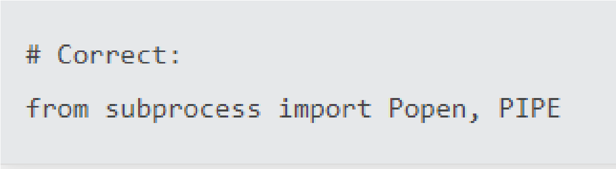
- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.
- Imports should be grouped in the following order:
  - Standard library imports.
  - Related third party imports.
  - Local application/library specific imports.
- You should put a blank line between each group of imports.
- Absolute imports are recommended, as they are usually more readable and tend to be better behaved
- Wildcard imports (from <module> import *) should be avoided



```
# Correct:
import os
import sys
```



```
# Wrong:
import sys, os
```

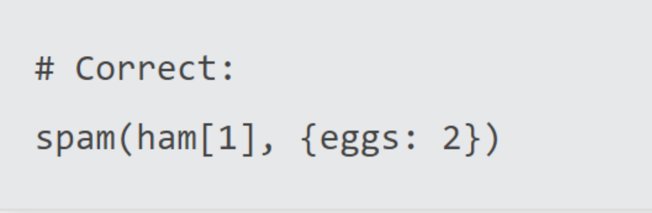


```
# Correct:
from subprocess import Popen, PIPE
```

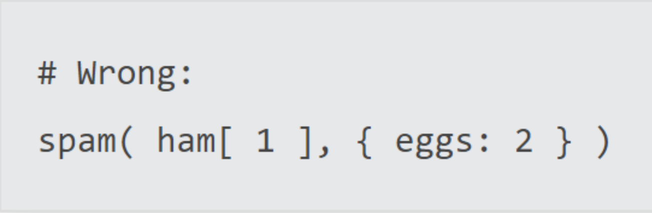
怎么漂亮怎么来

White Space: Pet Peeves (1)

- Avoid extraneous whitespace in the following situations:
  - Immediately inside parentheses, brackets or braces:

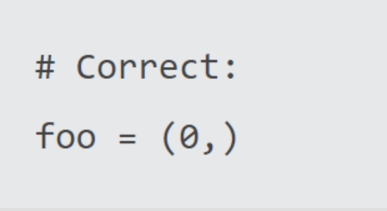


```
# Correct:
spam(ham[1], {eggs: 2})
```

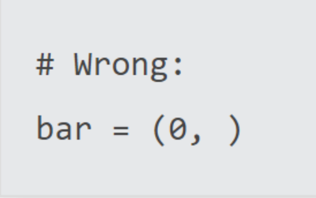


```
# Wrong:
spam( ham[ 1 ], { eggs: 2 } )
```

  - Between a trailing comma and a following close parenthesis:



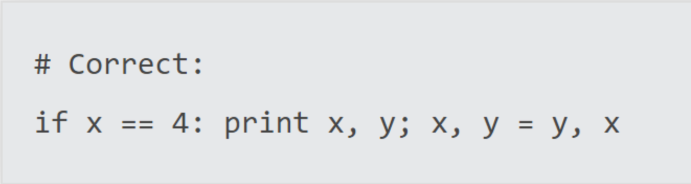
```
# Correct:
foo = (0,)
```



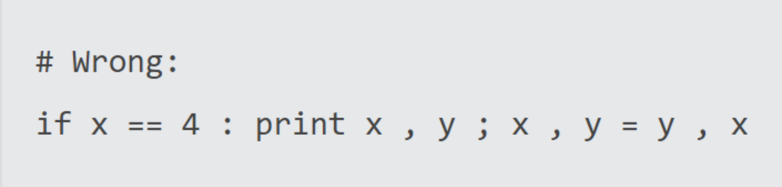
```
# Wrong:
bar = (0, )
```

White Space: Pet Peeves (2)

- Avoid extraneous whitespace in the following situations:
  - Immediately before a comma, semicolon, or colon:

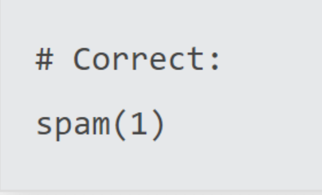


```
# Correct:
if x == 4: print x, y; x, y = y, x
```

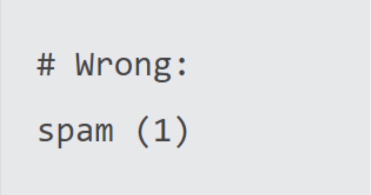


```
# Wrong:
if x == 4 : print x , y ; x , y = y , x
```

  - Immediately before the open parenthesis that starts the argument list of a function call:



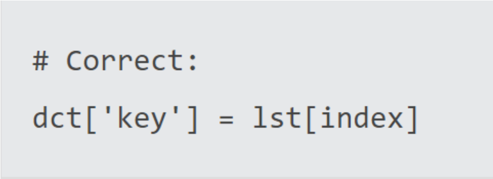
```
# Correct:
spam(1)
```



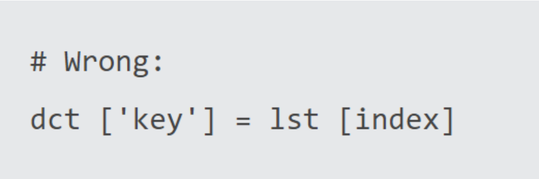
```
# Wrong:
spam (1)
```

White Space: Pet Peeves (3)

- Avoid extraneous whitespace in the following situations:
  - Immediately before the open parenthesis that starts an indexing or slicing:

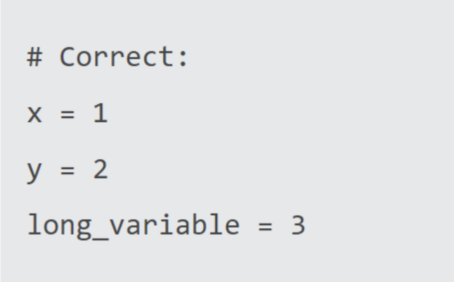


```
# Correct:
dct['key'] = lst[index]
```

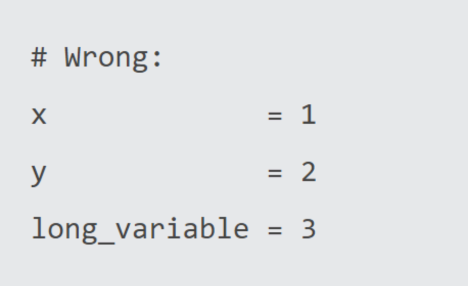


```
# Wrong:
dct ['key'] = lst [index]
```

  - More than one space around an assignment (or other) operator to align it with another:



```
# Correct:
x = 1
y = 2
long_variable = 3
```



```
# Wrong:
x             = 1
y             = 2
long_variable = 3
```

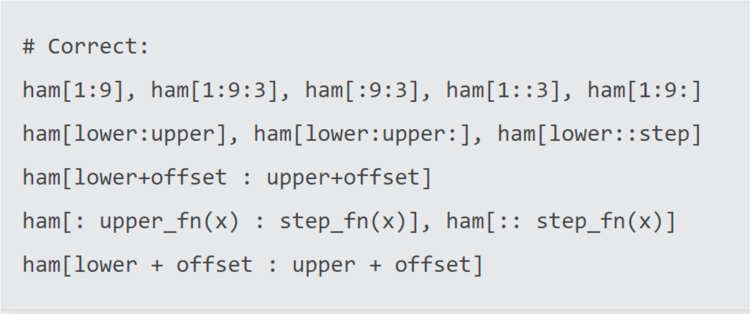
White Space: Recommendations

- Avoid trailing (结尾) whitespace anywhere.
- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator

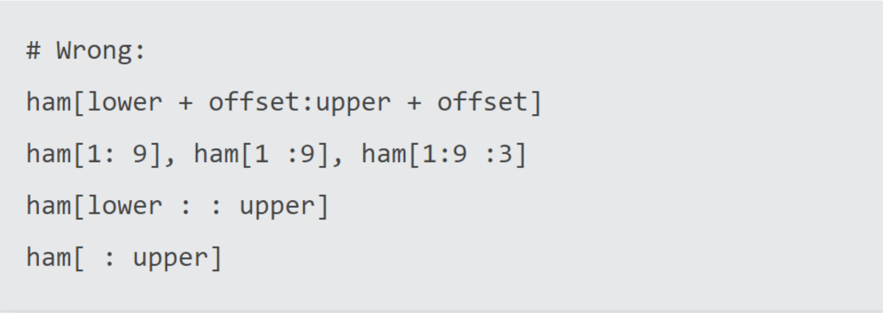
  $x{+}y{*}z$ $x + y{*}z$
- Function annotations should use the normal rules for colons and always have spaces around the -> arrow if present.
- Don't use spaces around the = sign when used to indicate a keyword argument, or when used to indicate a default value for an unannotated function parameter
  - When combining an argument annotation with a default value, however, do use spaces around the = sign:
- Compound statements (multiple statements on the same line) are generally discouraged
- While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

White Space: Summary

1. 代码符合英文写作规划
2. 用空格把程序切割成一个个合适的小的单元。每个单元有清晰的意思
3. 风格要统一
4. 不要挤成一坨
5. 清晰最重要



```
# Correct:
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]
```



```
# Wrong:
ham[lower + offset:upper + offset]
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower : : upper]
ham[ : upper]
```

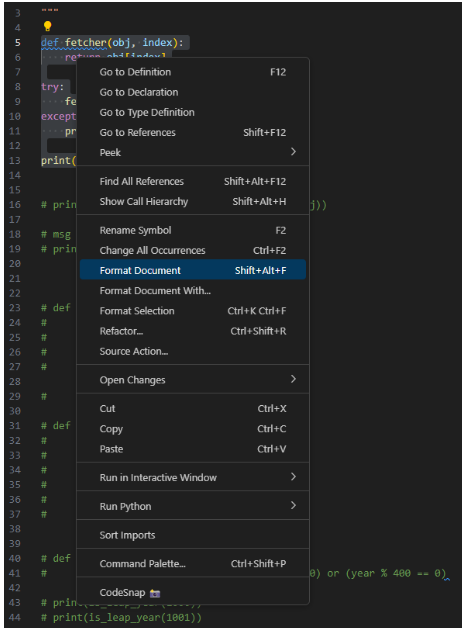
Name Convention

- Python命名: xxx_xxx_xxxxx
- 大小写区分规则： module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_CONSTANT_NAME, global_var_name, instance_var_name, function_parameter_name, local_var_name. CLASS_CONSTANT_NAME
- Names to Avoid
  - Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.
  - In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

https://www.python.org/dev/peps/pep-0008/

Useful VS Code Extensions

- autopep8, Black Format
- 解决很多格式问题
  - 空格
  - 布局
- Survey VS Code useful extensions



```
"""

def fetcher(obj, index):
    return obj[index]

try:
    fe
except
    pr

print(


# prin                                              j))

# msg
# prin



# def
#
#
#
#

#

# def
#
#
#
#
#
#


# def
#                                                   0) or (year % 400 == 0)

# prin
# print(is_leap_year(1001))
```

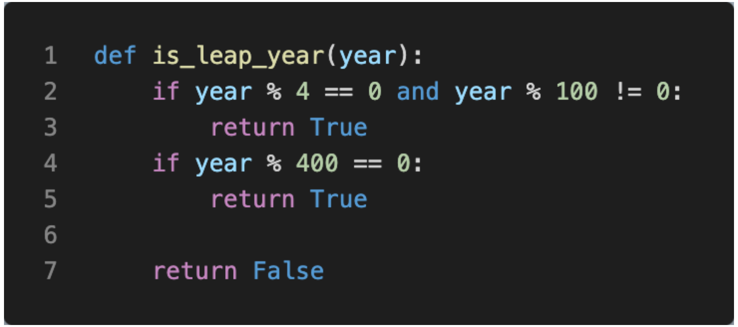
Go to Definition F12
Go to Declaration
Go to Type Definition
Go to References Shift+F12
Peek
Find All References Shift+Alt+F12
Show Call Hierarchy Shift+Alt+H
Rename Symbol F2
Change All Occurrences Ctrl+F2
Format Document Shift+Alt+F
Format Document With...
Format Selection Ctrl+K Ctrl+F
Refactor... Ctrl+Shift+R
Source Action...
Open Changes
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Run in Interactive Window
Run Python
Sort Imports
Command Palette... Ctrl+Shift+P
CodeSnap

Practice problems

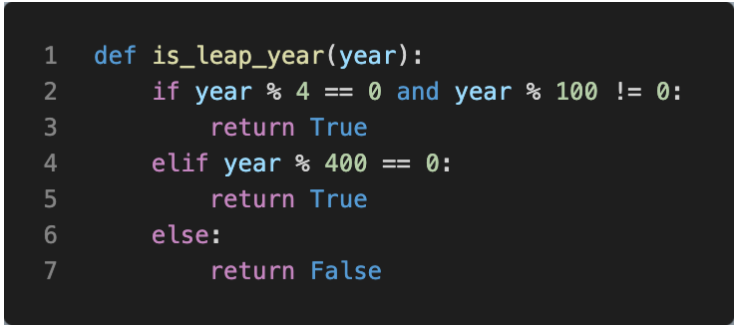
无他，唯手熟尔

Leap year

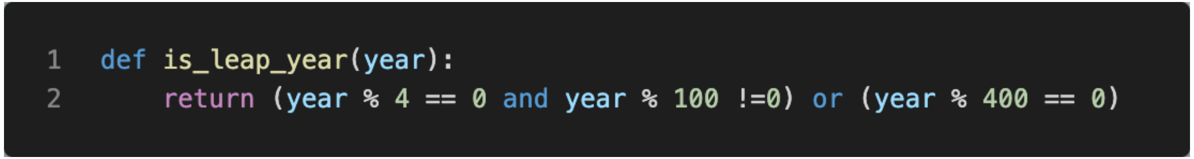
- A year is called leap:
  - It is divisible by 4 exactly
  - If it is divisible by 100, it should be divisible by 400
- Write a function to implement it, try to simplify your code



```python
def is_leap_year(year):
    if year % 4 == 0 and year % 100 != 0:
        return True
    if year % 400 == 0:
        return True

    return False
```



```python
def is_leap_year(year):
    if year % 4 == 0 and year % 100 != 0:
        return True
    elif year % 400 == 0:
        return True
    else:
        return False
```

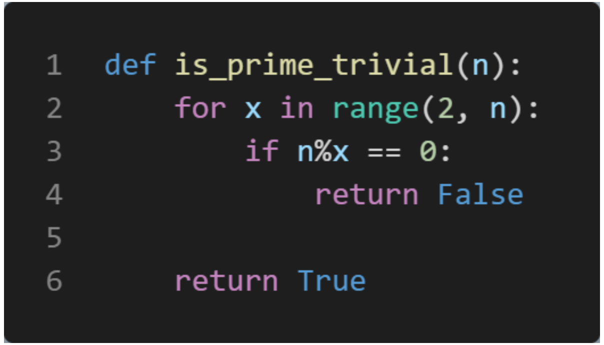


```python
def is_leap_year(year):
    return (year % 4 == 0 and year % 100 !=0) or (year % 400 == 0)
```

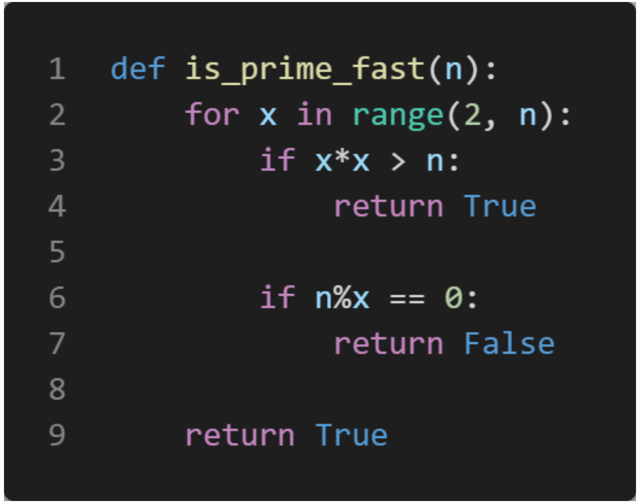
细节来源于积累

Prime Number

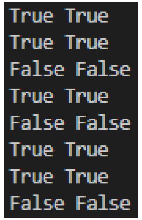
- A number n is prime is its only has divisors 1 and itself



```python
def is_prime_trivial(n):
    for x in range(2, n):
        if n%x == 0:
            return False

    return True
```



```python
def is_prime_fast(n):
    for x in range(2, n):
        if x*x > n:
            return True

        if n%x == 0:
            return False

    return True
```



```
True True
True True
False False
True True
False False
True True
True True
False False
```

Happy Number

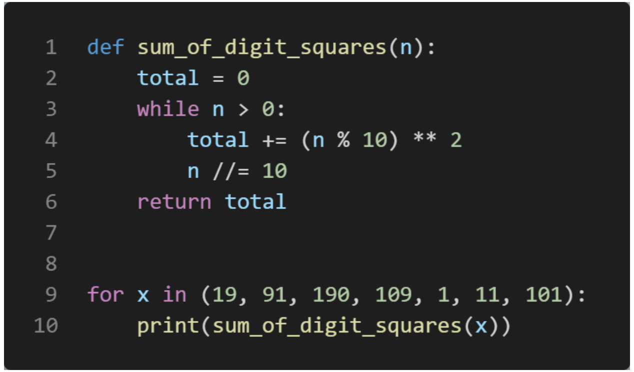
- (202-Happy Number) Write an algorithm to determine if a number is "happy".
- A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.
- Example: Input: 19 Output: true

Explanation:

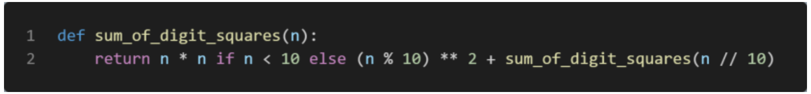
$$1^2 + 9^2 = 82\,; 8^2 + 2^2 = 68;\ 6^2 + 8^2 = 100; 1^2 + 0^2 + 0^2 = 1$$

- Analysis: $n \to f(n) \to f\big(f(n)\big) \to f^3(n) \ldots \to 1$ or Loop
    1. Implement a function $f(n)$: the sum of the squares of its digits
    2. How to check whether $1 \in S$? Data structure: list, tuple, dict, set. Dict or set?

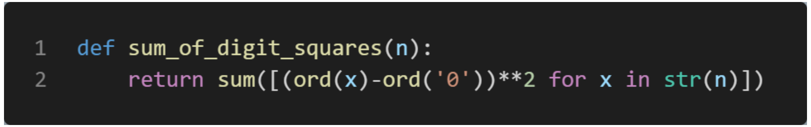
Happy Number: solution



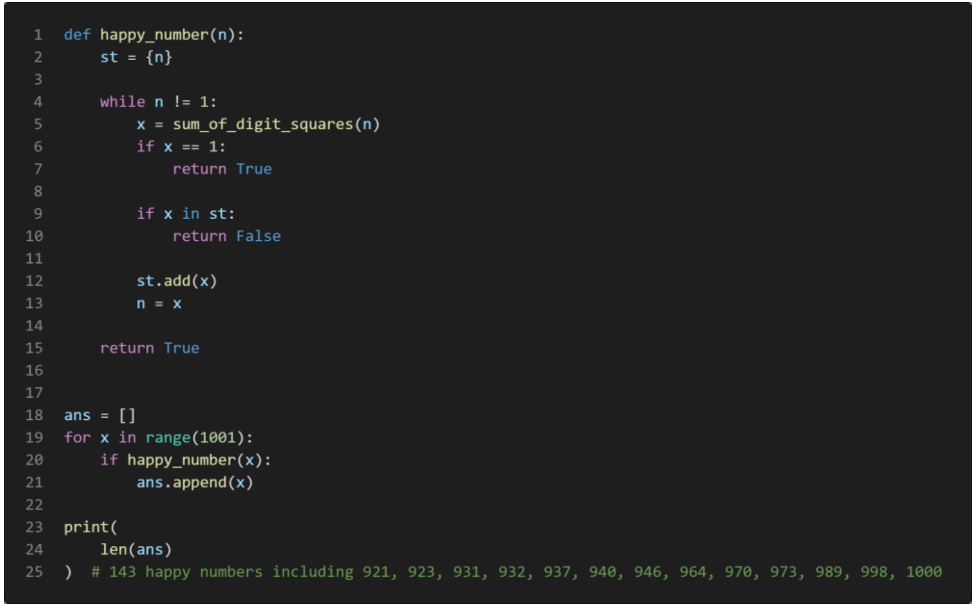
```python
def sum_of_digit_squares(n):
    total = 0
    while n > 0:
        total += (n % 10) ** 2
        n //= 10
    return total


for x in (19, 91, 190, 109, 1, 11, 101):
    print(sum_of_digit_squares(x))
```



```python
def sum_of_digit_squares(n):
    return n * n if n < 10 else (n % 10) ** 2 + sum_of_digit_squares(n // 10)
```



```python
def sum_of_digit_squares(n):
    return sum([(ord(x)-ord('0'))**2 for x in str(n)])
```

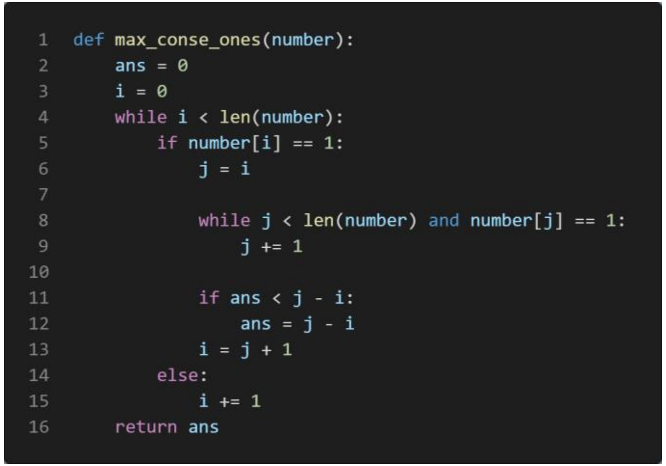


```python
def happy_number(n):
    st = {n}

    while n != 1:
        x = sum_of_digit_squares(n)
        if x == 1:
            return True

        if x in st:
            return False

        st.add(x)
        n = x

    return True


ans = []
for x in range(1001):
    if happy_number(x):
        ans.append(x)

print(
    len(ans)
)  # 143 happy numbers including 921, 923, 931, 932, 937, 940, 946, 964, 970, 973, 989, 998, 1000
```

不断改进

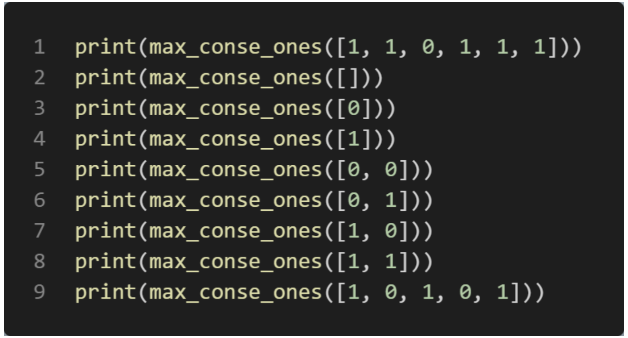
Max Consecutive Ones

- Given a binary array, find the maximum number of consecutive 1s in this array.
- Example 1: Input: [1, 1, 0, 1, 1, 1] Output: 3
- Explanation: The first two digits or the last three digits are consecutive 1s. The maximum number of consecutive 1s is 3.
- Note:
    - The input array will only contain 0 and 1.
    - The length of input array is a positive integer and will not exceed 10,000
- Analysis: how to check consecutive 1s.
    - Start from position i, i++ if the current position is 1

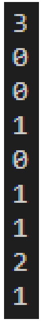


```python
def max_conse_ones(number):
    ans = 0
    i = 0
    while i < len(number):
        if number[i] == 1:
            j = i

            while j < len(number) and number[j] == 1:
                j += 1

            if ans < j - i:
                ans = j - i
            i = j + 1
        else:
            i += 1
    return ans
```

多测试



```python
print(max_conse_ones([1, 1, 0, 1, 1, 1]))
print(max_conse_ones([]))
print(max_conse_ones([0]))
print(max_conse_ones([1]))
print(max_conse_ones([0, 0]))
print(max_conse_ones([0, 1]))
print(max_conse_ones([1, 0]))
print(max_conse_ones([1, 1]))
print(max_conse_ones([1, 0, 1, 0, 1]))
```



```
3
0
0
1
0
1
1
2
1
```

Longest Continuous Increasing Subsequence

- Given an unsorted array of integers, find the length of longest continuous increasing subsequence
- Example 1:

  Input: [1,3,5,4,7]   Output: 3

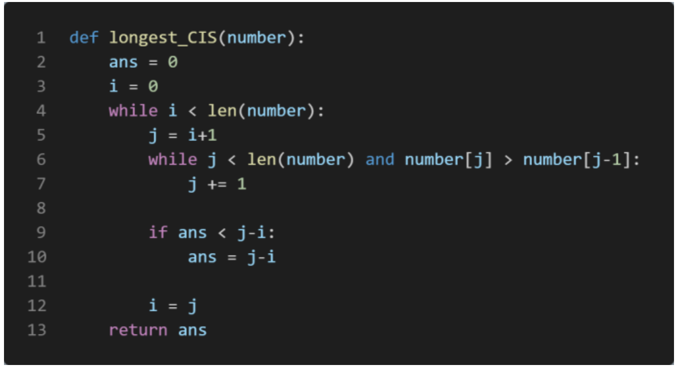
  Explanation: The longest continuous increasing subsequence is [1,3,5], its length is 3. Even though [1,3,5,7] is also an increasing subsequence, it's not a continuous one where 5 and 7 are separated by 4
- Example 2:

  Input: [2,2,2,2,2]   Output: 1

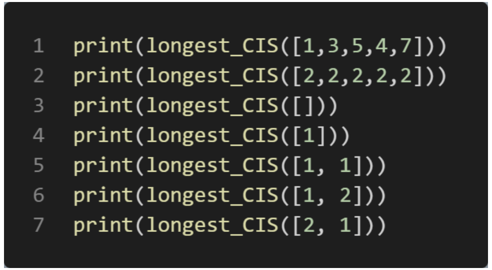
  Explanation: The longest continuous increasing subsequence is [2], its length is 1.

  Note: Length of the array will not exceed 10,000

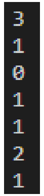
Analysis: very similar to Max Consecutive Ones



```python
def longest_CIS(number):
    ans = 0
    i = 0
    while i < len(number):
        j = i+1
        while j < len(number) and number[j] > number[j-1]:
            j += 1

        if ans < j-i:
            ans = j-i

        i = j
    return ans
```



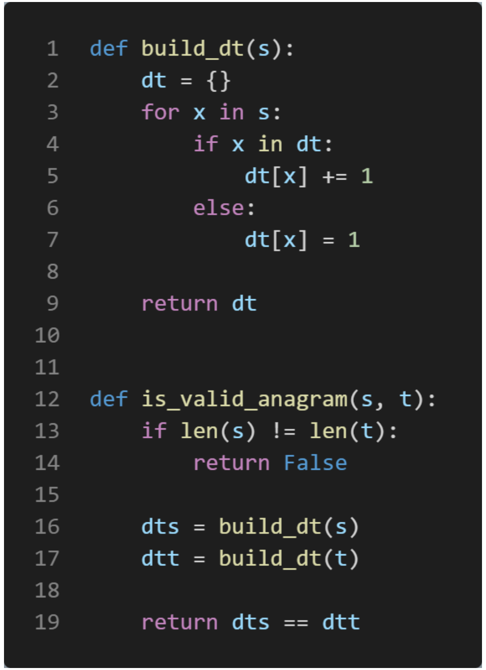
```python
print(longest_CIS([1,3,5,4,7]))
print(longest_CIS([2,2,2,2,2]))
print(longest_CIS([]))
print(longest_CIS([1]))
print(longest_CIS([1, 1]))
print(longest_CIS([1, 2]))
print(longest_CIS([2, 1]))
```



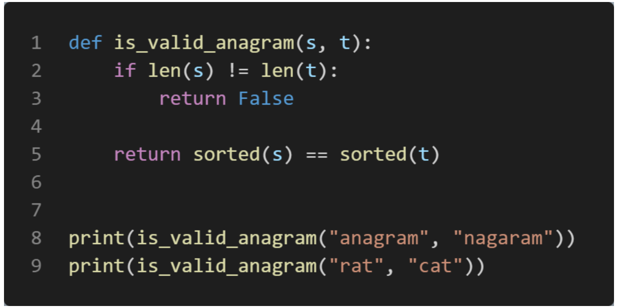
```
3
1
0
1
1
2
1
```

Valid Anagram

- (242-Valid Anagram) Given two strings s and t , write a function to determine if t is an anagram of s
    - An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once – race: care, part: trap, heart: earth, knee: knee
- Example 1: Input: s = "anagram", t = "nagaram" Output: true
- Example 2: Input: s = "rat", t = "car" Output: false
- Note: You may assume the string contains only lowercase alphabets
- Solution: every character should have the same occurrence in s and t
- Data structure: list, tuple, str, dict or set? Dict Vs. set?



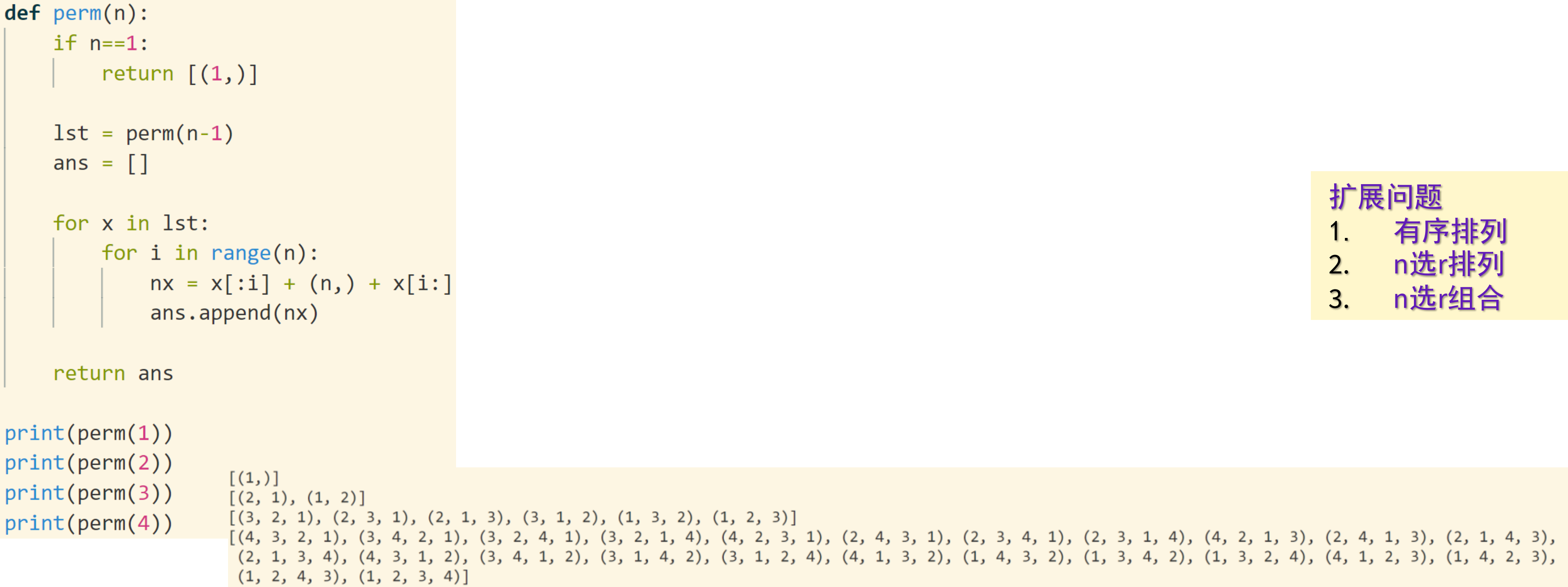
```python
def build_dt(s):
    dt = {}
    for x in s:
        if x in dt:
            dt[x] += 1
        else:
            dt[x] = 1

    return dt


def is_valid_anagram(s, t):
    if len(s) != len(t):
        return False

    dts = build_dt(s)
    dtt = build_dt(t)

    return dts == dtt
```



```python
def is_valid_anagram(s, t):
    if len(s) != len(t):
        return False

    return sorted(s) == sorted(t)


print(is_valid_anagram("anagram", "nagaram"))
print(is_valid_anagram("rat", "cat"))
```

permutation

- 问题：生成1,…,n的所有排列
- 问题具有递归的特点：1−n可以由1−(n−1)插入n得到
- 函数定义　　　　def perm(n): # return list：　每个元素是1-n的一个排列 [(), (), (),…]



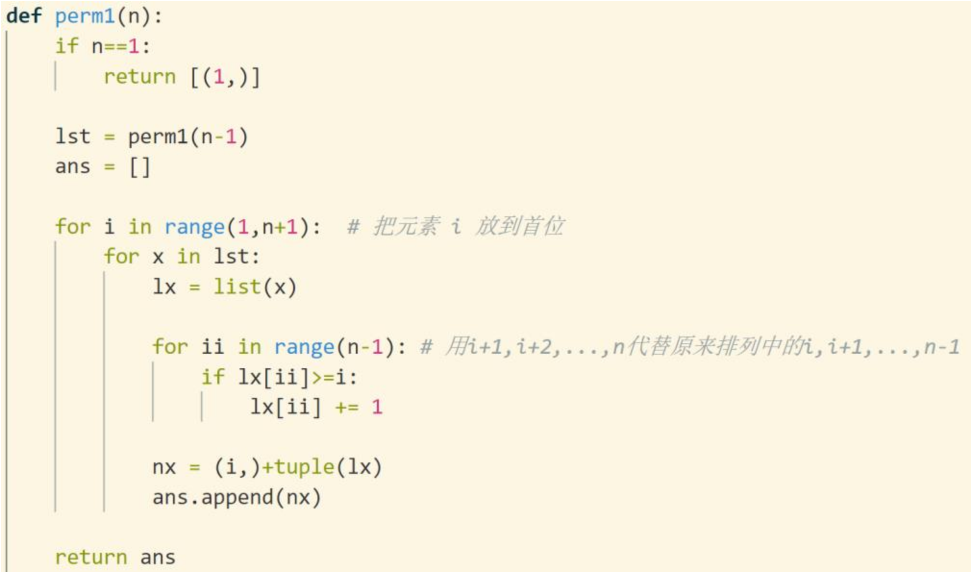
```python
def perm(n):
    if n==1:
        return [(1,)]

    lst = perm(n-1)
    ans = []

    for x in lst:
        for i in range(n):
            nx = x[:i] + (n,) + x[i:]
            ans.append(nx)

    return ans

print(perm(1))
print(perm(2))
print(perm(3))
print(perm(4))
```

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

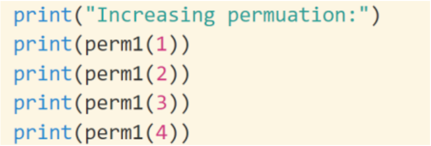
扩展问题

1. 有序排列
2. n选r排列
3. n选r组合

Increasing permutations



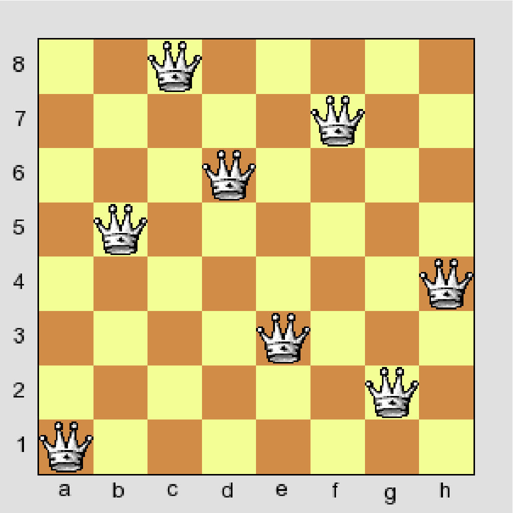
```python
def perm1(n):
    if n==1:
        return [(1,)]

    lst = perm1(n-1)
    ans = []

    for i in range(1,n+1):  # 把元素 i 放到首位
        for x in lst:
            lx = list(x)

            for ii in range(n-1): # 用i+1,i+2,...,n代替原来排列中的i,i+1,...,n-1
                if lx[ii]>=i:
                    lx[ii] += 1

            nx = (i,)+tuple(lx)
            ans.append(nx)

    return ans
```



```python
print("Increasing permuation:")
print(perm1(1))
print(perm1(2))
print(perm1(3))
print(perm1(4))
```

Eight Queens

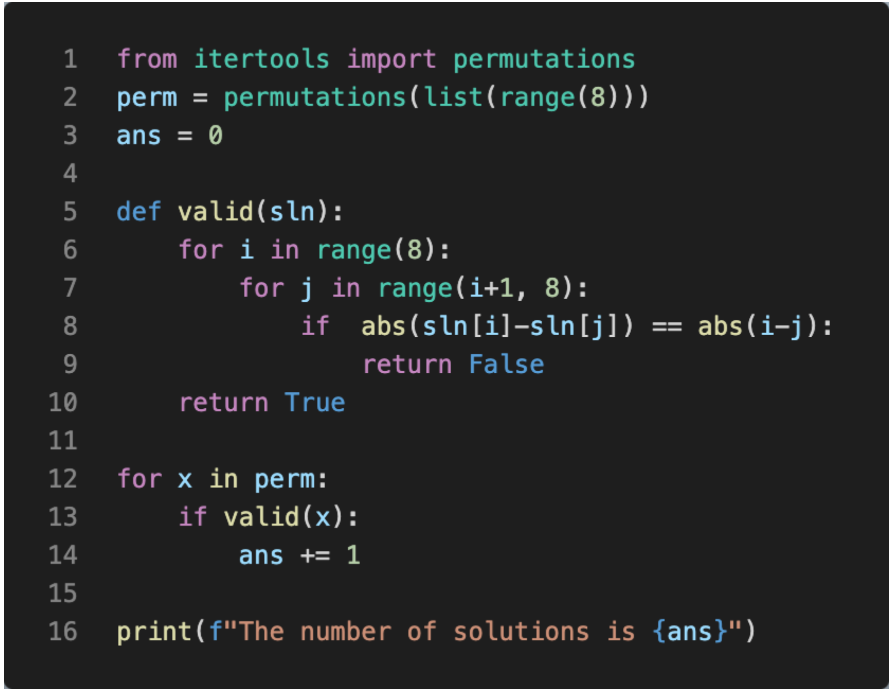
- Find the number of solutions for 8 Queens problem
  - How to check whether a given solution is valid
  - How to generate all the possible solutions
  - Ans: 92



如何表示一个正确的解
表示：用元组f表示放在各行的皇后的列号，那么f必须是1, ⋯, 8的一个排列。
思路：枚举1-8的所有排列，判断各个排列是不是合法的皇后放置方式（不同行、不同列，不同对角线）
最多8！种可能

Eight Queens: Code

- Python中自带了permutation函数，可以生成各种排列组合
- 本问题中，和前面的perm函数等价（比较测试你的代码是否正确）



```python
from itertools import permutations
perm = permutations(list(range(8)))
ans = 0

def valid(sln):
    for i in range(8):
        for j in range(i+1, 8):
            if  abs(sln[i]-sln[j]) == abs(i-j):
                return False
    return True

for x in perm:
    if valid(x):
        ans += 1

print(f"The number of solutions is {ans}")
```

规范：循环的迭代深度不超过2轮，
超过了用函数

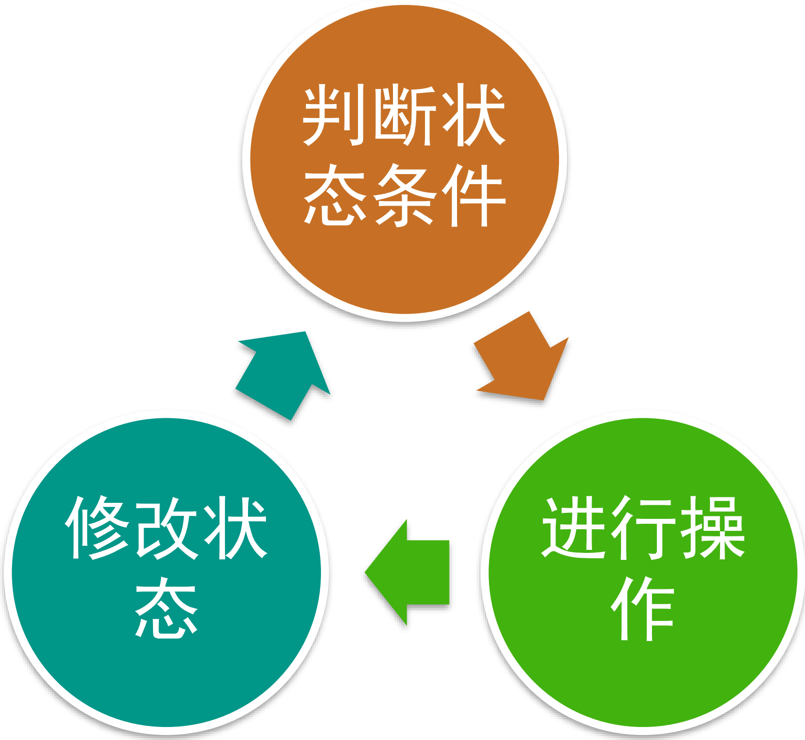
递归与循环

- 递归调用其实是个循环过程
- 递归函数调用需要额外时间、空间开销：传递参数，保存中间值，切换函数
  - 循环比函数递归更高效
- 递归函数更容易写，更符合人的思维模式
- 递归函数是把复杂问题转化为简单问题

  初学者滥用递归：能用递归的地方一律递归
- 循环是从简单条件出发一步步构造复杂情况
- Life is short, use python: Python给大家很多便利，更符合人的思维
- 计算机的思维：0/1
  - 你只能用“三角形盖房子”
  - 程序员来适应计算机
- 任何程序都可以用赋值、逻辑语句、判断语句、循环语句、跳转语句实现
- 难点：循环语句
  - while是一个复杂过程
- 如何通过设计一个循环来完成一个复杂的功能

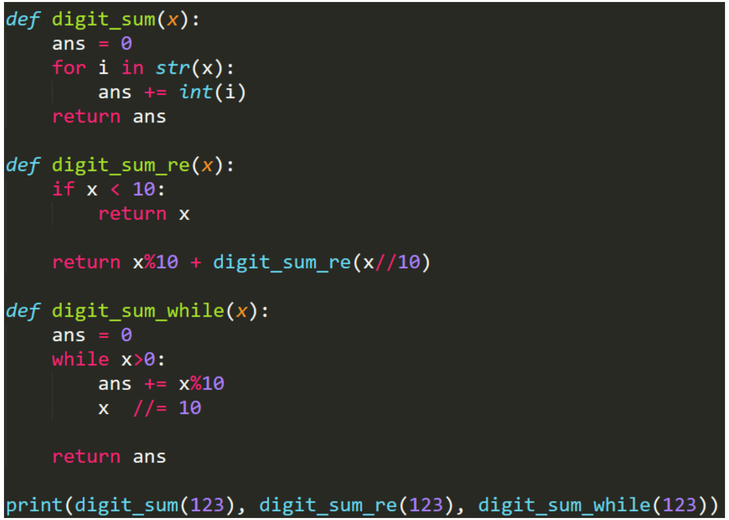
递归：从一般到特殊；循环：从特殊到一般

while: 状态更新

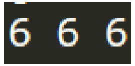


递归与循环

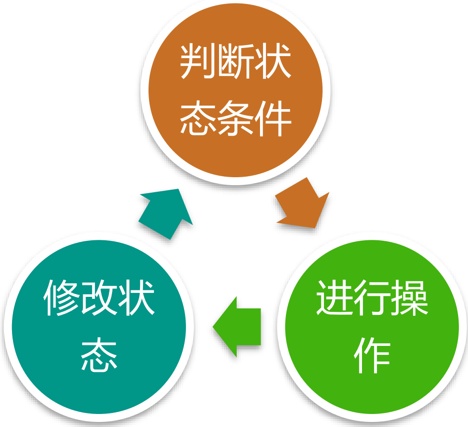
- 计算一个数各位数字的和



```python
def digit_sum(x):
    ans = 0
    for i in str(x):
        ans += int(i)
    return ans

def digit_sum_re(x):
    if x < 10:
        return x

    return x%10 + digit_sum_re(x//10)

def digit_sum_while(x):
    ans = 0
    while x>0:
        ans += x%10
        x  //= 10

    return ans

print(digit_sum(123), digit_sum_re(123), digit_sum_while(123))
```



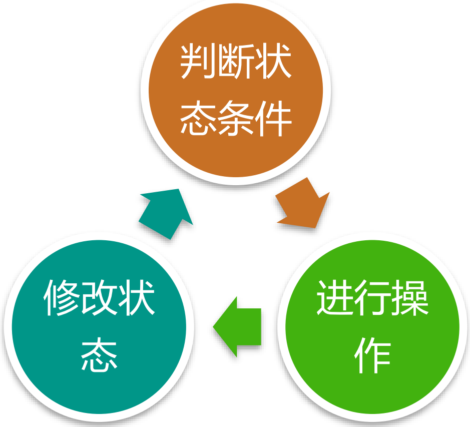
```
6 6 6
```

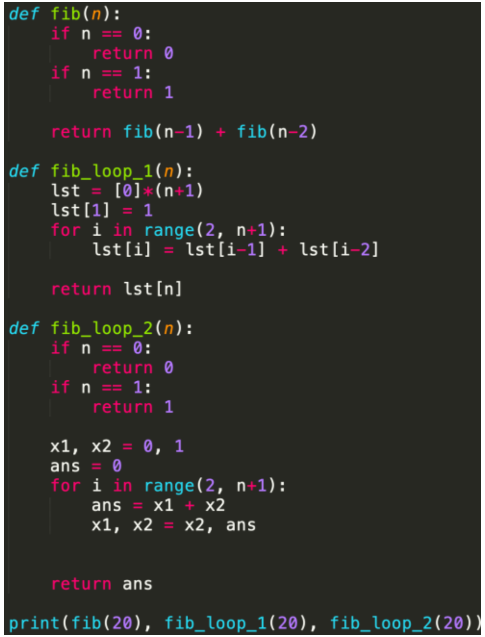


while: 必须明确地想清楚整个变化过程, 从i到i+1(递归函数自动完成)
并不是所有的递归都可以很轻松地用while写

递归与循环

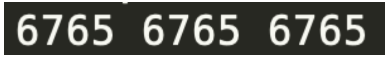
- 计算Fibonacci序列第n项





```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n-1) + fib(n-2)

def fib_loop_1(n):
    lst = [0]*(n+1)
    lst[1] = 1
    for i in range(2, n+1):
        lst[i] = lst[i-1] + lst[i-2]

    return lst[n]

def fib_loop_2(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    x1, x2 = 0, 1
    ans = 0
    for i in range(2, n+1):
        ans = x1 + x2
        x1, x2 = x2, ans


    return ans

print(fib(20), fib_loop_1(20), fib_loop_2(20))
```

- while: 必须明确地想清楚整个变化过程, 从i到i+1(递归函数自动完成)
- 并不是所有的递归都可以很轻松地用while写



```
6765 6765 6765
```

Recursive Functions → Loop

1. Write a Python program to calculate the sum of a list of numbers
2. Write a Python program to converting an integer to a string in any base.
3. Write a Python program of recursion list sum
    1. Test Data: [1, 2, [3,4], [5,6]]
    2. Expected Result: 21
4. Write a Python program to get the factorial of a non-negative integer
5. Write a Python program to get the sum of digitals of a non-negative integer
    1. Test Data:
    2. sumDigits(345) -> 12
    3. sumDigits(45) -> 9
6. Write a Python program to calculate the geometric sum of n items

    Note: In mathematics, a geometric series is a series with a constant ratio between successive terms

    Example :

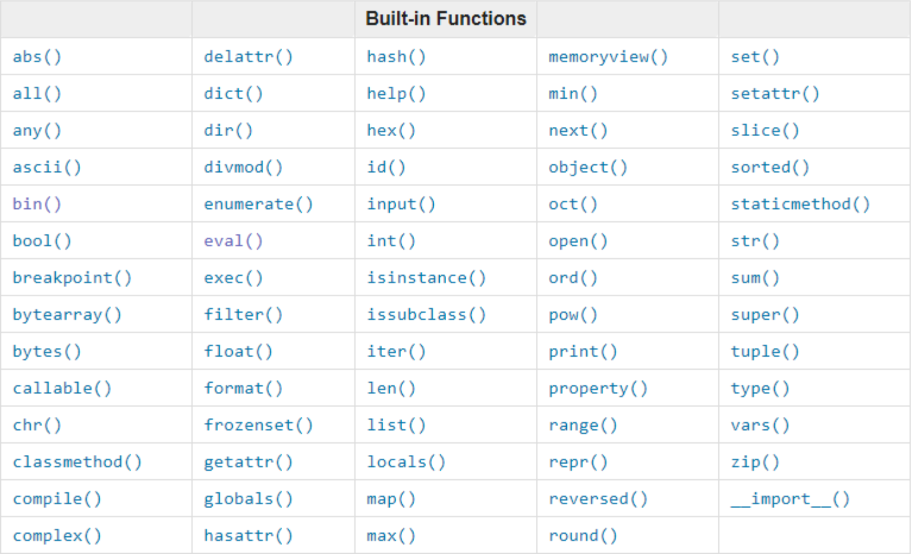
$$\sum_{i=0}^{n} ap^{i}$$

7. Write a Python program to calculate the value of 'a' to the power 'b’

    Test Data : power(3,4) -> 81
8. Fibonacci, gcd, climbing steps, binary search

Built-in functions

- The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order



		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()	

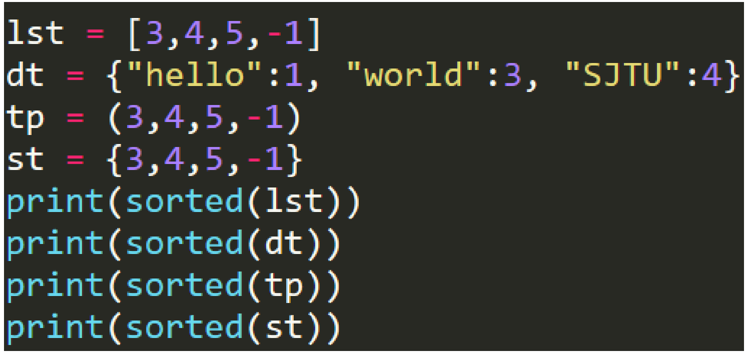
https://docs.python.org/3.8/library/functions.html

Tips: 当你需要实现一个基本的功能的时候，python可能已经帮你实现了

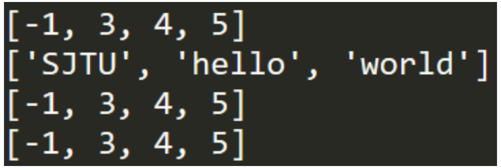
sorted: 排序

Python lists have a built-in list.sort() method that modifies the list in-place. There is also a sorted() built-in function that builds a new sorted list from an iterable

- sorted(x): 对x排序并return新的list变量
- list.sort(): 对列表排序，无返回值
  - https://docs.python.org/3/howto/sorting.html



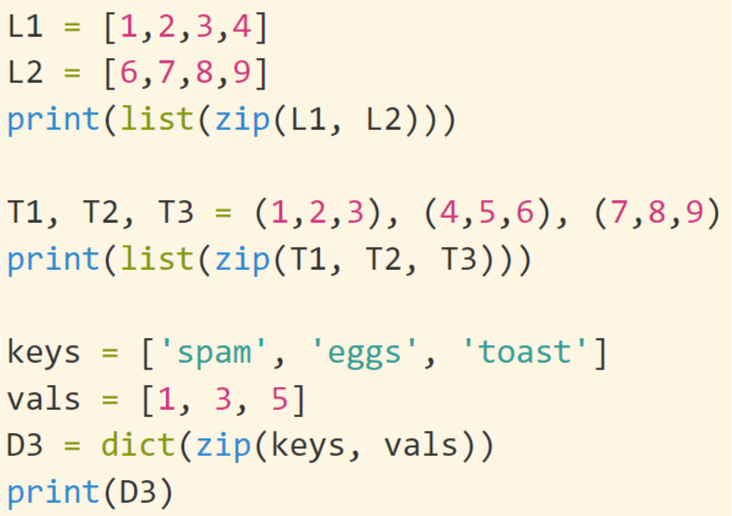
```
lst = [3,4,5,-1]
dt = {"hello":1, "world":3, "SJTU":4}
tp = (3,4,5,-1)
st = {3,4,5,-1}
print(sorted(lst))
print(sorted(dt))
print(sorted(tp))
print(sorted(st))
```



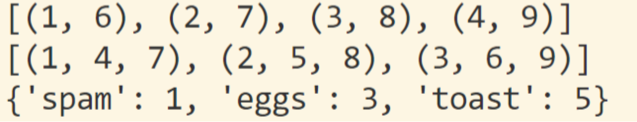
```
[-1, 3, 4, 5]
['SJTU', 'hello', 'world']
[-1, 3, 4, 5]
[-1, 3, 4, 5]
```

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



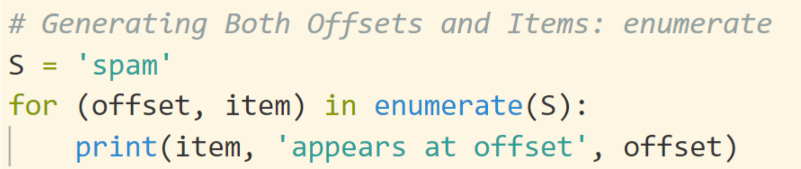
```
L1 = [1,2,3,4]
L2 = [6,7,8,9]
print(list(zip(L1, L2)))

T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
print(list(zip(T1, T2, T3)))

keys = ['spam', 'eggs', 'toast']
vals = [1, 3, 5]
D3 = dict(zip(keys, vals))
print(D3)
```



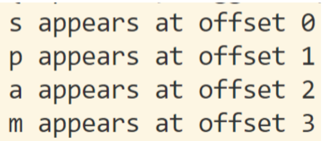
```
[(1, 6), (2, 7), (3, 8), (4, 9)]
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
{'spam': 1, 'eggs': 3, 'toast': 5}
```

enumerate()

- Generating both offsets（元素的序号） and items: enumerate



```python
# Generating Both Offsets and Items: enumerate
S = 'spam'
for (offset, item) in enumerate(S):
    print(item, 'appears at offset', offset)
```



```
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

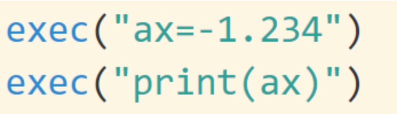
eval()

- eval可以用来计算一个字符串形式的表达式的值
  - eval(“123+456”)
  - 返回值是表达式的值
- eval完整的功能要强大很多: It is an interesting hack/utility in Python which lets a Python program run Python code within itself
  - The eval() method parses the expression passed to it and runs python expression(code) within the program.
- 譬如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]]”)
- 计算一般的表达式
  - x= 1 y =2 z=eval(“(x+1)**y+y”)
  - program = input('Enter a program:')
  - eval(program) #[print(item) for item in [1, 2, 3]]
- Error: eval(“a=1”)

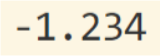
https://www.programiz.com/python-programming/methods/built-in/eval
https://www.geeksforgeeks.org/eval-in-python/

exec()

- The exec() method executes the dynamically created program, which is either a string or a code object
    - Return none
- program = input('Enter a program:’)
  exec(program) #[print(item) for item in [1, 2, 3]]



```
exec("ax=-1.234")
exec("print(ax)")
```



```
-1.234
```

https://www.programiz.com/python-programming/methods/built-in/exec
https://stackoverflow.com/questions/2220699/whats-the-difference-between-eval-exec-and-compile