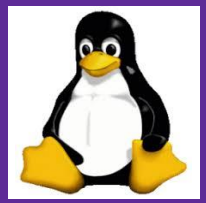
Introduction to Computation

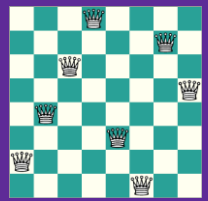
Autumn, 2023

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
  - 注意边界条件：: x=[], “”, (,), 0, None
  - 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

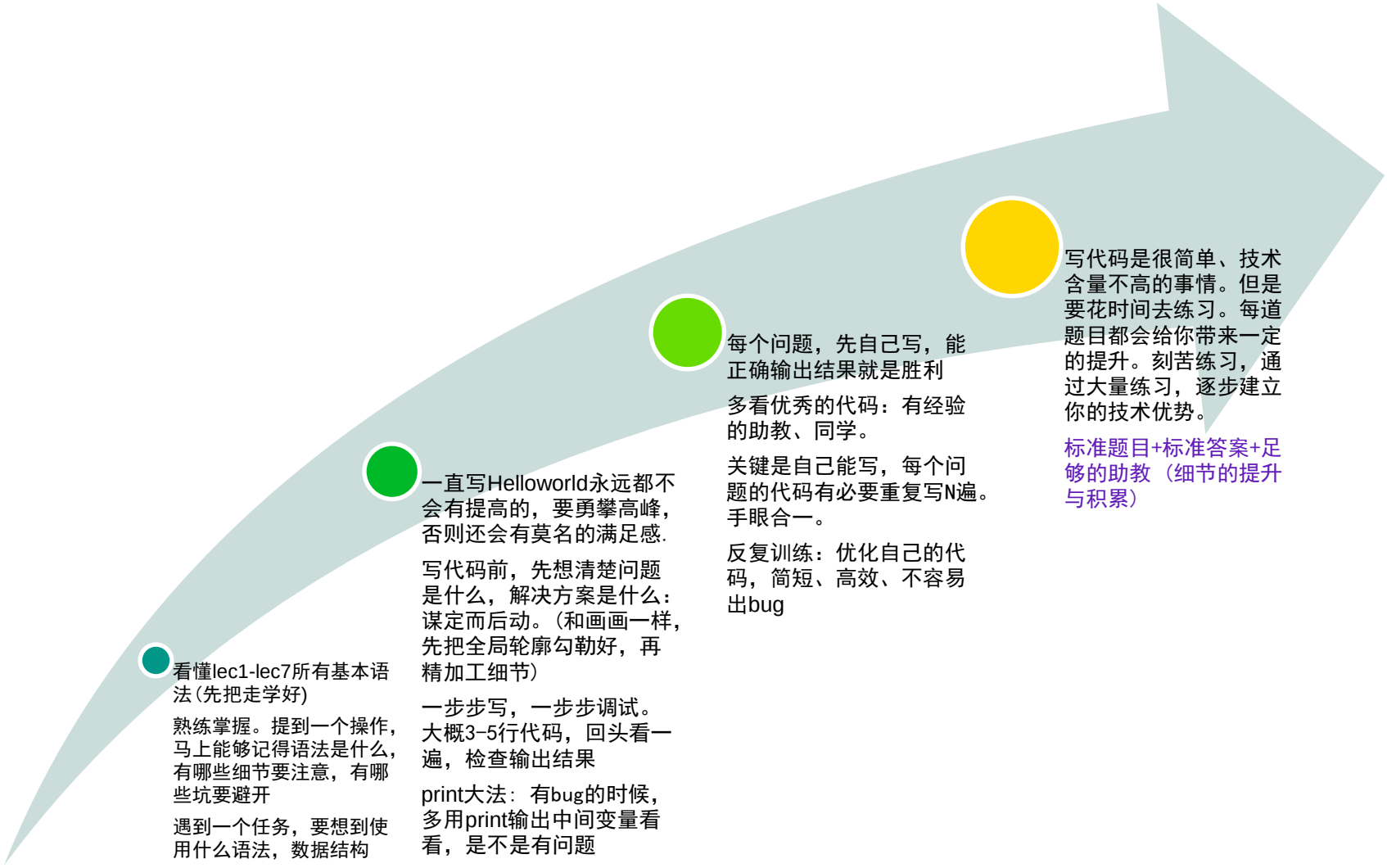
Common Errors

- TypeError
- NameError
- None
- SyntaxError
- IndentationError
- AttributeError
- ZeroDivisionError
- 测试每个函数、每种可能情况
  - 注意特殊输入，譬如n=0，字符串为""，列表为[]，字典为{}
- Debug：强烈建议print大法，在每个关键步骤后输出变量信息

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
  - 本课程只是让大家大量的练习语法，多了解CS的问题
- 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

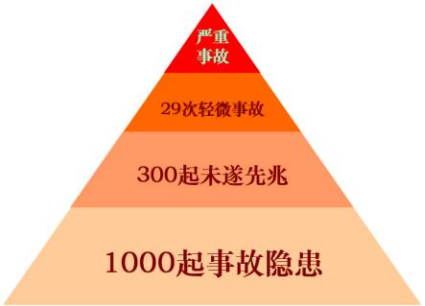
不贪多，每天保证3-5道就足够了
每道题务必做透
刚开始速度慢，一天3题，熟练了，可以远远超过3题
2个月的练习到期末前大概可以做完200道


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



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
```

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]
```

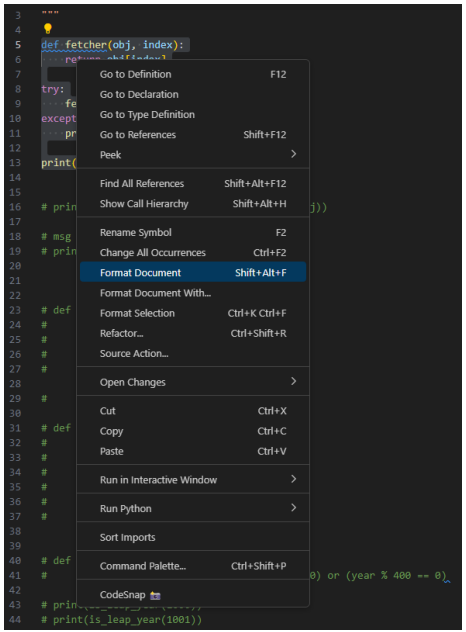
Name Convention

- 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



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

代码技巧

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


# 等价于
def f(n):
    return n == 1
```

```python
def f(n):
    if g(n):
        return a(n)
    else:
        return b(n)


# 等价于
def f(n):
    return a(n) if g(n) else b(n)
```

```python
# Bad
if v == True:
    f()

# Good
if v:
    f()

# Bad
if v == False:
    f()

# Good
if not v:
    f()
```

① 能够连写的地方一律连写
x = x +1
x += 1
② 函数定义内部不要用print，一律用return。除非题目让你print。调试可以用print
③ 调试的时候用小规模数据，不要一开始就1,000,000. 可以从1，2，3，4，5，6开始
④ 问问题记得截屏，不要手机拍照，发源代码，发图
⑤ 作业题不要用超出范围的语法，我们确保可以用讲过的语法解决
⑥ 不包含注释，单个函数的代码长度不要超过40行。超过这个长度，要么切割成更小的几个几个函数，要么你的思路有问题，要重新写
⑦ 务必先熟练掌握ppt上面的内容，做到能自己写出来
⑧ 编程是一个精密的数学工程，不要用蛮力，不要暴力复制粘贴

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: $n \to f(n) \to f\big(f(n)\big) \to f^3(n)$ ... $\to 1$ 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
```

```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
```

多测试

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"))
```