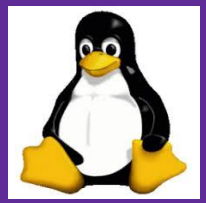
Introduction to Computation

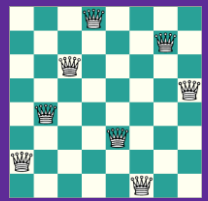
Autumn, 2023

Prof. Fan Cheng

Shanghai Jiao Tong University

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





11

Outline

- Algorithm complexity
- Exception
- Assertion

Algorithm complexity

Correctness and Performance

- The most important thing to think about when designing and implementing a program is that it should produce results that can be relied upon.
- Sometimes performance is an important aspect of correctness. This is most obvious for programs that need to run in real time. A program that warns airplanes of potential obstructions needs to issue the warning before the obstructions are encountered.

Time: measure performance

- In Python, we can use the time module to study the running time of a program: https://docs.python.org/3/library/time.html
- We could run the program on some input and time it. But that wouldn't be particularly informative because the result would depend upon
  - the speed of the computer on which it is run
  - the efficiency of the Python implementation on that machine
  - the value of the input
- The time depends on the machine and system. How to measure its running time in theory?
- It is cumbersome to use time.time() or timeit.timeit()
- We need a theory to predict the future

```python
import time

def f(n):
    answer = 1
    while n >= 1:
        answer *= n
        n -= 1

    return answer

t1 = time.time()

n = 3000
rounds = 100

for i in range(rounds):
        res = f(n)

t2 = time.time()
print(t1, t2, (t2-t1)/rounds)
```

Nothing is practical than a good theory

```
1605497847.6290567 1605497847.8165107 0.001874539852142334
```

How to measure performance

- **Idea:** computer programs consists of fundamental instructions like
  - +, -, *, /, //, %, if, =, print, input……
  - Regarded the running time of these instructions as “1”
  - For each program, just count how many fundamental instructions it contains
- A step is an operation that takes a fixed amount of time, such as binding a variable to an object, making a comparison, executing an arithmetic operation, or accessing an object in memory

```python
print("Hello World")  # one step
print("This is an apple")  # one step
for i in range(n):  # n step
    print(i)

a = b + c  # one step
a /= d  # one step
print(a**d)  # one step


def fib(n):  # 2**n steps
    if n == 0:
        return 1
    if n == 1:
        return 1

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

the number of steps may depend on the input

Worst-case Running Time

- Both L and x will affect the running time
- **Best-case running time**: the minimum running time over all the possible inputs of a given size
- **Worst-case running time**: the maximum running time over all the possible inputs of a given size
- **Average-case running time**: the average running time over all possible inputs of a given size

```
def linear_search(L, x):
    for e in L:
        if e == x:
            return True

    return False
```

Asymptotic Notation

- Observed that, when n is large,
  - 5n and 5n+2 are very close
  - $1000000n$ and n^2 are very far away
- We are concerned with n is very large in theory: the order（阶数） is more important
- asymptotic notation: the order of an algorithm
  - The underlying motivation is that almost any algorithm is sufficiently efficient when run on small inputs
  - What we typically need to worry about is the efficiency of the algorithm when run on very large inputs. As a proxy for “very large,” asymptotic notation describes the complexity of an algorithm as the size of its inputs approaches infinity

“Big O” notation

- The order of complexity is much more important
- The asymptotic complexity of an algorithm:
  - If the running time is the sum of multiple terms, keep the one with the largest growth rate, and drop the others
  - If the remaining term is a product, **drop any constants**
- The most commonly used asymptotic notation is called “Big O” notation. Big O notation is used to give an upper bound on the asymptotic growth (often called the order of growth) of a function
  - **O(1)** denotes constant running time
  - **O($\log n$)** denotes logarithmic running time
  - **O(n)** denotes linear running time
  - **O($n \log n$)** denotes log-linear running time
  - **O(n^k)** denotes polynomial running time. Notice that k is a constant
  - **O(c^n)** denotes exponential running time. Here a constant is being raised to a power based on the size of the input

Asymptotic Notation: Summary

- Not real time consumed by algorithms
- Time approximated by steps.
- If the size of input is large enough, asymptotic time complexity is accurate to compare the time complexity of algorithms
- Understand time complexity is the foundation of computer science and engineering
- From now on, when you write a program, you should be Very Very sensitive to the time complexity of your algorithm

Complexity in Python statement

- 要根据操作数是数, 字符串判断或者是自定义类型判断
- Python的基本数据类型没有范围限制，可以为无穷长
- 小规模的数可以看作O(1): 小规模 -2^{64}~ 2^{64}；如果超过这个范围，要特别对待
- 字符串和数一样，短字符串可以看做O(1)

- =, +, -, *, /, //, %, +=, -/=…, **
- >=, >, <=, <, ==, !=
- not
- if…elif…else
- print(), input(): 取决于输入类型的长度. 一般是O(1)
- math.sin(), math.cos()
- return

- 复杂语法由其定义决定
  - while()
  - for()
  - 自定义类、函数def f():

High Precision (高精度)

- 遇到python的+, - 等要注意整数的大小，才能确定复杂度
- Operating system: 32 bits, 64bits
- 一般情况下，一个整数的范围 $-2^{32} \sim 2^{32}$ $-2^{64} \sim 2^{64}$
- 溢出：整数操作超过了上述范围
- Python整数没有限制，$\pm 2^{100000}$都可以表示
- 列表模拟高精度：
  - 用一个列表表示一个任意长度的整数，列表的每一位对应于整数相应位置的数字
  - 两个任意长度整数的叫法，通过列表来完成：模拟对位相加，进位等等
  - 可以模拟+, -, *, /, // 等等

- 对位相加
- 进位
- 最高位进位

```
  987
+ 123
 1110
```

```python
def add(x, y):
    z = [0]*(max(len(x), len(y))+1)

    for i in range(len(z)):
        if i<len(x):
            z[i] += x[i]
        if i<len(y):
            z[i] += y[i]

    for i in range(len(z)-1):
        z[i+1] += z[i]//10
        z[i] %= 10

    if z[len(z)-1]==0:
        del z[len(z)-1]

    return z
```

```python
def str2list(x):
    lst = [0]*len(x)
    for i in range(len(x)):
        lst[i] = int(x[len(x)-1-i])

    return lst
```

```python
xs = "12345678909876543 21"
ys = "98765432101234567890"
x = str2list(xs)
y = str2list(ys)
print(x)
print(y)
print(add(x,y))

xs = "12"
ys = "99"
x = str2list(xs)
y = str2list(ys)
print(x)
print(y)
print(add(x,y))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[0, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9]
[2, 1]
[9, 9]
[1, 1, 1]
```

- 用字符串表示一个大整数

pow(), **

- a^x: pow(a, x), a**x内部实现一样
  - x 如果是整数, $O(\log x)$：二分法
  - 如果a是很大的整数，要考虑a的长度
- pow(a, x, n)数学上等于 $a ** x\%n$
  - pow(a, x, n)比$a ** x\%n$会快, 尤其是a, x很大的情况 **(高精度会影响速度)**
  - 原因：$a\%n$始终$\leq n$, 而$a ** x$可能是一个非常大的整数，最后的答案一定是在$[0, n-1]$

```python
def my_pow(a, x):
    if x == 0:
        return 1

    t = my_pow(a, x // 2)

    return t * t if x % 2 == 0 else t * t * a


print(my_pow(2, 0))
print(my_pow(2, 1))
print(my_pow(2, 2))
print(my_pow(2, 7))
print(my_pow(2, 10))
print(my_pow(2, 20))
print(my_pow(2, 200))
```

```
1
2
4
128
1024
1048576
1606938044258990275541962092341162602522202993782792835301376
```

Evaluate a Polynomial

- $f_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$
- Directly computation
  - Time complexity $O(n\log n)$
- Horner's rule
  - $f_n(x) = x(a_n x^{n-1} + \cdots + a_2 x + a_1) + a_0 = x f_{n-1}(x) + a_0$
  - $f_0(x) = a_n$
  - Time complexity $O(n)$

列表的复杂度

- 列表的基本假设
  - 列表中的元素在内存中是依次连续排列的
  - 列表中的元素是通过下标访问的: lst[i]
    - 访问一个元素是O(1)
- 在这两个假设下
  - 插入 O(n)
  - 查找 O(n)
  - 删除 O(n)
- 所有的操作都要保证，经过操作后的数据结构依然是列表，即满足两个基本假设

- 依次连续排列才能保证通过下标访问
- 下标和元素一一对应

100	2	3	4	5	7	8	-1	-3	-7

列表的复杂度

- 在基本假设下，如何自己实现插入、查找、删除
- 插入 O(n)

100	2	3	4	5	7	8	-1	-3	-7	
100	2	3	4		5	7	8	-1	-3	-7
100	2	3	4	6	5	7	8	-1	-3	-7

- 查找 O(n)

100	2	3	4	5	7	8	-1	-3	-7

- 删除 O(n)

100	2	3	4	5	7	8	-1	-3	-7
100	2	3	4	5	*	8	-1	-3	-7
100	2	3	4	5	8	-1	-3	-7	

- 修改和插入很费时
- 不要一边遍历，一边修改

列表的复杂度

- 给一个1024长度的内存，模拟insert, remove, find. (用list模拟内存，list[i]第i个内存)

```python
def insert(lst, x, pos):
    global nlst
    for i in range(nlst, pos, -1):
        lst[i] = lst[i-1]

    lst[pos] = x
    nlst += 1


def find(lst, x):
    for i in range(nlst):
        if lst[i] == x:
            return i

    return -1

def remove(lst, x):
    global nlst
    for i in range(nlst):
        if lst[i] == x:
            for j in range(i, nlst-1):
                lst[j] = lst[j+1]
            nlst -= 1
            return
```

```python
lst = [1,2,3,4,5,6] + ['*']*1018 #内存大小为1024
nlst = 6 # 实际长度 *表示空位置

print(lst[:nlst])
insert(lst, 100, 3)
print(lst[:nlst])
insert(lst, 200, 5)
print(lst[:nlst])

print(find(lst, 100))

print(lst[:nlst])
remove(lst, 5)
print(lst[:nlst])
remove(lst, 100)
print(lst[:nlst])
remove(lst, 6)
print(lst[:nlst])
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 100, 4, 5, 6]
[1, 2, 3, 100, 4, 200, 5, 6]
3
[1, 2, 3, 100, 4, 200, 5, 6]
[1, 2, 3, 100, 4, 200, 6]
[1, 2, 3, 4, 200, 6]
[1, 2, 3, 4, 200]
```

复合数据结构list, tuple, str, dict, set

- list, tuple, str: 内存中，数据依次(consecutive)存储，排列成一行
  - list=[1, 2, 3, …, n]: 1 2 3 … n
    - 如果列表嵌套、列表含有不同类型，需要单独分析
  - tuple=(1, 2, 3): 1 2 3 … n
  - str="abc…z": a b c … z
- dict, set: 内部实现是hash table，需要更多的额外空间
- 插入、删除：
  - list, tuple, str: $O(n)$ (tuple, str不能修改，而是建立一个新的变量)
    - 删掉后，列表中不能有空位，要把所有后面的元素往前移一位
    - 插入时，要把所有后面的元素往后移一位，然后插进去
  - dict, set: $O(1)$
- 查找：x in collection_a
  - list, tuple, str: $O(n)$
  - dict, set: $O(1)$
- List.append(): $O(1)$

Hash (哈希) Table

-2		3	
	100		
			-8
		7	

- 预先准备一个表(连续分布的一块内存空间)：需要时间和空间
- 对每个元素x, 用表中一个位置$pos(x)$来存储
- $pos(x)$叫做hash函数，需要自己设计（难点）
- 理想状况 $pox(x) \neq pos(y), \forall x \neq y$

- A new idea for store and search data, take string processing for example
- Assume we have a large table to store data:

  length = 100001, hash_table = [0]*length
- For each word w, use a function f to compute a position pos to store w

  pos = f(w), hash_table[pos] = w
  - The computation of function f should be linear: O(|w|)
  - There should be no or very few collision（hash碰撞）: $pos(w_1) \neq pos(w_2), w_1 \neq w_2$
- Then, we could insert, find and delete each word w in $O(|W|) \rightarrow O(1)$.

Hash (哈希) Table

-2		3	
	100		
			-8
		7	

- 预先准备一个表：需要时间和空间
- 对每个元素x, 用表中一个位置pos(x)来存储

- The function f is called a hash function (hash： 混乱无章)
- 哈希函数的设计是有技巧的：为了避免冲突
  - table的length要足够大，越大越不容易碰撞（空间需求大）
  - 用空间换时间：O(1)来源于更大的空间
  - 有碰撞可以解决（本课程不深入）
- 常用hash函数

```python
length = 10001
hash_table = [0] * length
name_table = [""] * length


def pos(key):
    ans = 0
    for x in key:
        ans = (ans * 255 + ord(x)) % length

    return ans


lst = speech.split()

for word in lst:
    word = word.lower()
    pword = pos(word)
    if not hash_table[pword]:
        hash_table[pword] = 1
        name_table[pword] = word
    else:
        hash_table[pword] += 1

for i, x in enumerate(hash_table):
    if hash_table[i] >= 3:
        print(f"{name_table[i]}: {hash_table[i]}")
```

- 例子：实际中通过位置函数pos(x)来存储、查询
  - 电影院直接告诉你座位在m排n列
  - 宴会上会告诉你在第几桌几号位
- 不是顺序比较来查找

- Hash函数pos(x)需要设计。假定字符都是ASCII码，即小于256，这里的pos(key)的设计，是把字符串看做一个256进制的整数
- Hash函数的设计是一个很复杂的问题
  - No pain no gain

- List: 插入、查找、删除都是O(n)
- dict(), set(): 插入、查找、删除都是 O(1)
- Hash table 的问题：初始化需要额外时间和空间
  - 查找、插入、删除：O(1)
  - Python: __hash__()
  - 尽量只初始化使用一个dict, set

hash()

- hash(object)
  - Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).
  - For objects with custom __hash__() methods, note that hash() truncates the return value based on the bit width of the host machine. See __hash__() for details.

```
6760262601136192184
-1341123136733738793
1896
1051464412201451643
0.30000000000000004
False
False
```

```python
def test_hash(x):
    print(hash(x))


test_hash("hello world")
test_hash("SJTU")
test_hash(1896)
test_hash(123.456)

x = 0.1
print(x + x + x)
print(x + x + x == 0.3)
print(hash(3 * x) == hash(0.3))
```

Example I

```
def int_to_str(i):
    """Assumes i is a nonnegative int
    Returns a decimal string representation of i"""
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i%10] + result
        i = i//10
    return result
```

Example II

```
def add_digits(s):
    """Assumes s is a str each character of which is a
    decimal digit.
    Returns an int that is the sum of the digits in s"""
    val = 0
    for c in s:
        val += int(c)
    return val
```

Example III

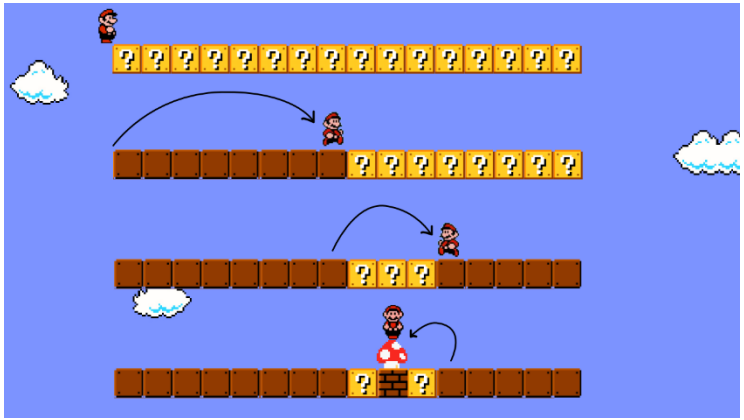
```
def intersect(L1, L2):
    """Assumes: L1 and L2 are lists
    Returns a list that is the intersection of L1 and L2"""
    #Build a list containing common elements
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
     #Build a list without duplicates
     result = []
     for e in tmp:
         if e not in result:
             result.append(e)
      return result
```

Example IV

```
def is_subset(L1, L2):
    """Assumes L1 and L2 are lists.
    Returns True if each element in L1 is also in L2
    and False otherwise."""
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

Binary Search

- 对于从小到大排好序的列表，每次和中间的元素比较，可以排除掉一半的元素



Sorted list, Binary search: O(log n)
in : O(n)

```python
def binary_search(lst, x, h, t):
    if h > t:
        return -1
    if lst[(h + t) // 2] == x:
        return (h + t) // 2
    elif lst[(h + t) // 2] > x:
        return binary_search(lst, x, h, (h + t) // 2 - 1)
    else:
        return binary_search(lst, x, (h + t) // 2 + 1, t)


li = [1, 3, 5, 6, 23, 65, 100, 1000]

print(binary_search(li, 23, 0, len(li) - 1))
print(binary_search(li, 1, 0, len(li) - 1))
print(binary_search(li, 1000, 0, len(li) - 1))
print(binary_search(li, -23, 0, len(li) - 1))
print(binary_search(li, 2113, 0, len(li) - 1))
print(binary_search(li, 70, 0, len(li) - 1))
```

```
4
0
7
-1
-1
-1
```

- 注意+1， -1
- 用while实现

Example VI: Fibonacci sequence

- $Fib(n+2) = Fib(n+1) + Fib(n)$:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

- Time complexity ≈$O(2^n)$
  - $T(n+2) = T(n+1) + T(n)$
  - $T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$
  - 每个$Fib(n)$都被重复计算

Example VI: Speed Up Fibo.

- Tips: use a list (or dict) lst to store the intermediate results to speed up the computation
  - Each item in lst is initialized as -1
  - When invoke f(n), we could check lst[n] to see whether lst[n] had been computed before
    - If so, just return f(n)
    - Otherwise, we invoke the recursive function to compute f(n) and store it in lst[n]

```python
fib = [-1] * 10000
fib[0], fib[1], fib[2] = 0, 1, 1

def memorized_fibonacci(n):
    if n<=2:
        return fib[n]

    if fib[n] != -1:
        return fib[n]

    f1 = memorized_fibonacci(n-1)
    f2 = memorized_fibonacci(n-2)
    fib[n] = f1+f2

    return fib[n]

print("Fibonacci n = 1")
print(memorized_fibonacci(1))
print("Fibonacci n = 7")
print(memorized_fibonacci(7))
print("Fibonacci n = 800")
print(memorized_fibonacci(800))
```

```
Fibonacci n = 1
1
Fibonacci n = 7
13
Fibonacci n = 800
69283081864224717136290077681328518273399124385204820718966040597691435587278383112277161967532530675374170
85740474301762346722036177801617210685583897575985190398725
```

```python
dt = {0:0, 1:1, 2:1}
def fast_fib(n):
    if n in dt:
        return dt[n]

    dt[n] = fast_fib(n-1) + fast_fib(n-2)
    return dt[n]

print("Fast Fib: ", fast_fib(900))
print(timeit.timeit('fast_fib(800)', globals=globals(), number=1))
```

```
Fast Fib:  54877108839480000051413673948383714443800519309123592724494953427039811201064341234954387521525390615 5
04949092187441218246679104731442473022013980160407007017175697317900483275246652938800
6.000000007944095e-07
```

Python cache mechanism

```python
import functools
import sys
print(sys.getrecursionlimit())
sys.setrecursionlimit(3000)

@functools.lru_cache(maxsize=8192)
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)

print("Fibonacci with cache:")
print(fibonacci(800))
```

仅供了解 ☺

```
1000
Fibonacci with cache:
69283081864224717136290077681328518273399124385204820718966040597691435587278383112277161967532530675374170
85740474301762346722036177801617210685583897575998519039872 5
```

Exercise

- Fibonacci序列
- gcd
- 合并两个list
- 合并两个排序好的list
- 测试list, set的效率
- 从时间复杂度可以类似的定义空间复杂度

Exception

Even grammar is correct

三者的平衡：用户、程序员、Python解释器

- x * (y/z)
- >>> 10 * (1/0)

  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ZeroDivisionError: division by zero
- x + y*3
- >>> 4 + spam*3

  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError: name 'spam' is not defined

- x + 2
- >>> '2' + 2

  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: Can't convert 'int' object to str implicitly

It is not an error. It is an Exception
Your code is correct. Users do not follow your instructions.

Exception: solution

- Quadratic formula $f(x) = ax^2 + bx + c, (a \neq 0)$
- We assume that $a \neq 0$, but….

```
7  def quadratic(a, b, c):
8
9      delta = b*b - 4*a*c
10
11     return (-b + delta**0.5)/(2*a),  (-b - delta**0.5)/(2*a)
12
13
14 def quadratic1(a, b, c):
15
16     delta = b*b - 4*a*c
17
18     if a == 0.0:
19         print("error")
20         return
21
22     return (-b + delta**0.5)/(2*a),  (-b - delta**0.5)/(2*a)
```

Another example

- Windows C++ programming
- ReadFromFile(file_name):

- In our mind,
- filename = "C:\\Helloworld\\1.txt"
- ret = ReadFromFile("C:\\Helloworld\\1.txt")

- In practice, we need to
  - Check whether the file exist
  - Check whether the file can be opened
  - Check whether the return value ret is valid

- It is too distracting. I only want to read from a file!

干扰了正常的业务逻辑

Exception

Errors detected during execution are called exceptions and are not unconditionally fatal

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Most exceptions are not handled by programs, however, and result in error messages as shown here
- The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message:
  - the types in the example are ZeroDivisionError, NameError and TypeError
- Built-in exceptions:
  - https://docs.python.org/3/library/exceptions.html#bltin-exceptions

Handling Exception: Try except

- Two keywords **try** and **except**
  - First, the try clause (the statement(s) between the try and except keywords) is executed. If no exception occurs, the except clause is skipped and execution of the try statement is finished
  - If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement
  - If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above

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

Exercise

- Find potential sources of runtime errors in this code snippet:
  - dividend = float(input("Please enter the dividend: "))
  - divisor = float(input("Please enter the divisor: "))
  - quotient = dividend / divisor
  - quotient_rounded = math.round(quotient)
- for x in range(a, b):

  print("({},{}, {})“.format(my_list[x]))
- Add a try-except statement to the body of this function which handles a possible **IndexError**, which could occur if the index provided exceeds the length of the list. Print an error message if this happens:

```
def print_list_element(thelist, index):
    print(thelist[index])
```

Raise

The raise statement allows the programmer to force a specified exception to occur.

- >>> raise NameError('HiThere')
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError: HiThere
- >>>raise ZeroDivisionError()
- >>>raise ZeroDivisionError
- >>>raise TypeError
- >>>raise TypeError()
- >>>raise ValueError
- If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
1  try:
2      raise NameError("HiThere")
3  except NameError:
4      print("An exception flew by!")
5      raise  # try # raise?
```

```
An exception flew by!
Traceback (most recent call last):
  File "C:\Users\fcheng\OneDrive\CS124计算导论\2018\lecture notes\test_exception.py", line 3, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

Clean up: try … finally(终极)

- The try statement has another optional clause which is intended to define clean-up actions that **must be executed under all circumstances.**

```
try:
    raise KeyboardInterrupt
finally:
    print('Goodbye, world!')
```

```
Goodbye, world!
Traceback (most recent call last
  File "C:\Users\fcheng\OneDrive
    raise KeyboardInterrupt
KeyboardInterrupt
```

- A finally clause is always executed before leaving the try statement, whether an exception has occurred or not.
- When an exception has occurred in the try clause and has not been handled by an except clause (or it has occurred in an except or else clause), it is re-raised after the finally clause has been executed.
- The finally clause is also executed “on the way out” when any other clause of the try statement is left via a **break, continue or return** statement.
- As you can see, the finally clause is executed in any event. The TypeError raised by dividing two strings is not handled by the except clause and therefore re-raised after the finally clause has been executed.
- In real world applications, the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

```
def test_try_finally():
    try:
        return 'from_try'
    finally:
        return 'from_finally'

print(test_try_finally())
```

Exception: else

- The try … except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.
- The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try … except statement.
- **else** will be executed only if the try clause doesn't raise an exception

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
divide(2, 1)
divide(2, 0)
divide("2", "1")
```

```
result is 2.0
executing finally clause
division by zero!
executing finally clause
executing finally clause
Traceback (most recent call last):
  File "C:\Users\fcheng\OneDrive\CS124计算导论\2018\lecture no
    divide("2", "1")
  File "C:\Users\fcheng\OneDrive\CS124计算导论\2018\lecture no
    result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Predefined Clean-up Actions

- Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
print(line, end="")
```

- The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications.

- The **with statement** allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
for line in f:
print(line, end="")
```

- After the statement is executed, the file f is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation

More on try and except

- A try statement may have more than one except clause, to specify handlers for different exceptions.
- At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement.
- An except clause may name multiple exceptions as a parenthesized tuple, for example:
  ... except (RuntimeError, TypeError, NameError):
  ... pass #do nothing
- The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well)

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

User Defined Exception

仅供了解 ☺

- Programs may name their own exceptions by creating a new exception class (see Classes for more about Python classes).
- Exceptions should typically be derived from the Exception class, either directly or indirectly.
- Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.
- When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```
class Error(Exception):
    """Base class for exceptions in this
    module."""
    pass
class InputError(Error):
    """Exception raised for errors in the
    input.
    Attributes:
        expression -- input expression in
    which the error occurred
        message -- explanation of the error
    """
def __init__(self, expression, message):
        self.expression = expression
        self.message = message
```

```
class TransitionError(Error):
    """Raised when an operation attempts a
    state transition that's not  allowed.
    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific
        transition is not allowed
    """
    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Error checks vs exception handling

```
#with checks

n = None
while n is None:
    s = input("Please enter an integer: ")
    if s.lstrip('-').isdigit():
        n = int(s)
    else:
        print("%s is not an integer." % s)
```

```
# with exception handling

n = None
while n is None:
    try:
        s = input("Please enter an integer: ")
        n = int(s)
    except ValueError:
        print("%s is not an integer." % s)
```

Exercise

- This function adds an element to a list inside a dict of lists. Rewrite it to use a try-except statement which handles a possible KeyError if the list with the name provided doesn't exist in the dictionary yet, instead of checking beforehand whether it does. Include else and finally clauses in your try-except block:

```
def add_to_list_in_dict(thedict, listname, element):
    if listname in thedict:
        l = thedict[listname]
        print("%s already has %d elements." % (listname, len(l)))
    else:
        thedict[listname] = []
        print("Created %s." % listname)

    thedict[listname].append(element)
    print("Added %s to %s." % (element, listname))
```

Application

- How to tell whether a given string is a number?
  - “123”, “-3.14”, “-100”
  - “123a”, ”345.888+”

```
44 def is_number(s):
45     try:
46         float(s)
47         return True
48     except ValueError:
49         return False
50
51 print(is_number("123"))
52 print(is_number("3.14"))
53 print(is_number("+123"))
54 print(is_number("-3.14"))
55 print(is_number("123.a"))
56 print(is_number("-12a3"))
57 print(is_number("123+123"))
58 print(is_number("12..456"))
```

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

Assertion

assert

assert expression[, assertion_message]

- Sanity checks in your program:
  if x<0: print(“error”)
- The assert keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError
  - if not (0<x<1): print(“x is not valid”)
  - assert 0<x<1
  - assert r==0
- assertion_message is optional
- Logic Error: assert (condition, message)
  - (condition, message) is a tuple
  - A tuple like (condition, message) is always True
  - Remove the () above
  - Multiline: \

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


test_assertion(0.1)
test_assertion(0.2)
test_assertion(0.3)
# (test_assertion(0)) # AssertionError
test_assertion(1) #AssertionError
```

```
r=0.1 is valid.
r=0.2 is valid.
r=0.3 is valid.
Traceback (most recent call last):
  File "/Users/fancheng/Library/CloudStorage/OneDrive-Personal/CS124计算导论/2023 秋季/course_code.py", line 21, in <module>
    test_assertion(1) #AssertionError
  File "/Users/fancheng/Library/CloudStorage/OneDrive-Personal/CS124计算导论/2023 秋季/course_code.py", line 13, in test_assertion
    assert 0 < r < 1, f"{r=} is expected to be between 0 and 1."
AssertionError: r=1 is expected to be between 0 and 1.
```

```python
r = 0.4
assert 0 < r < 1, \
    f"{r=} is expected to be between 0 and 1."
```

Assertion Use Cases

- Python's assert statement allows you to write sanity checks in your code.
  - These checks are known as assertions, and you can use them to test if certain assumptions remain true while you're developing your code.
  - If any of your assertions turn false, then you have a bug in your code
- Assertions are a convenient tool for documenting, debugging, and testing code during development.
  - Once you've debugged and tested your code with the help of assertions, then you can turn them off to optimize the code for production.
  - Assertions will help you make your code more efficient, robust, and reliable
- The assertion condition should always be true unless you have a bug in your program
- In general, you shouldn't use assertions for data processing or data validation, because you can disable assertions in your production code, which ends up removing all your assertion-based processing and validation code
  - Using assertions for data processing and validation is a common pitfall
- Additionally, assertions aren't an error-handling tool
  - The ultimate purpose of assertions isn't to handle errors in production but to notify you during development so that you can fix them

隔离正常的业务逻辑

https://realpython.com/python-assert-statement/

Documenting Your Code With Assertions

- To alert programmers to this buggy call, you can use a comment
- However, using an assert statement can be more effective:
  - After the condition test, your code stops running, so it avoids abnormal behaviors and points you directly to the specific problem
- Using assertions is an effective and powerful way to document your intentions and avoid hard-to-find bugs due to accidental errors or malicious actors

```
1  def get_response(server, ports=(443, 80)):
2      # The ports argument expects a non-empty tuple
3      for port in ports:
4          if server.connect(port):
5              return server.get()
6      return None
```

```
1  def get_response(server, ports=(443, 80)):
2      assert len(ports) > 0, \
3          f"ports expected a non-empty tuple, got {ports}"
4      for port in ports:
5          if server.connect(port):
6              return server.get()
7      return None
```

Bug is not Exception

- To properly use assertions as a debugging tool, you shouldn't use try … except blocks that catch and handle AssertionError exceptions.
  - If an assertion fails, then your program should crash because a condition that was supposed to be true became false.
  - You shouldn't change this intended behavior by catching the exception with a try … except block.
- A proper use of assertions is to inform developers about unrecoverable errors in a program.
  - Assertions shouldn't signal an expected error, like a FileNotFoundError, where a user can take a corrective action and try again.
- The goal of assertion should be to uncover programmers' errors rather than users' errors.
  - Assertions are useful during the development process, not during production.
  - By the time you release your code, it should be (mostly) free of bugs and shouldn't require the assertions to work correctly.

Except： 用户错误
Assert：程序员错误

Disabling Assertions in Production for Performance

Assertions are great during development, but in production, they can affect the code's performance.

- Now say that you've come to the end of your development cycle. Your code has been extensively reviewed and tested. All your assertions pass, and your code is ready for a new release. At this point, you can optimize the code for production by disabling the assertions that you added during development
- two options:
- Run Python with the -O or -OO options.
    - Set the PYTHONOPTIMIZE environment variable to an appropriate value.
- Python has a built-in constant called __debug__.
    - This constant is closely related to the assert statement.
    - Python's __debug__ is a Boolean constant, which defaults to True.
    - It's a constant because you can't change its value once your Python interpreter is running:

```
1  import builtins
2
3  print("__debug__" in dir(builtins))
4  print(__debug__)
```

__debug__

```
1   __debug__ = False
```

- True is the default value of __debug__, and there's no way to change this value once your Python interpreter is running
- The value of __debug__ depends on which mode Python runs in, normal or optimized:
  - Normal mode is typically the mode that you use during development, while optimized mode is what you should use in production
- Now, what does __debug__ have to do with assertions? In Python, the assert statement is equivalent to the following conditional:
  - If __debug__ is true, then the code under the outer if statement runs.
  - The inner if statement checks expression for truthiness and raises an AssertionError only if the expression is not true. T
  - his is the default or normal Python mode, in which all your assertions are enabled because __debug__ is True

```
1  if __debug__:
2      if not expression:
3          raise AssertionError(assertion_message)
4
5  # Equivalent to
6  assert expression, assertion_message
```

__debug__ (cont'd)

- On the other hand, if __debug__ is False, then the code under the outer if statement doesn't run, meaning that your assertions will be disabled.
  - In this case, Python is running in optimized mode
- Normal or debug mode allows you to have assertions in place as you develop and test the code.
  - Once your current development cycle is complete, then you can switch to optimized mode and disable the assertions to get your code ready for production
- To activate optimized mode and disable your assertions, you can either start up the Python interpreter with the –O or -OO option, or set the system variable PYTHONOPTIMIZE to an appropriate value
  - The -O option internally sets __debug__ to False.
  - This change removes the assert statements and any code that you've explicitly introduced under a conditional targeting __debug__.
  - The -OO option does the same as -O and also discards docstrings.

Optimized Mode

- When you run Python, the interpreter compiles any imported module to bytecode on the fly
- The compiled bytecode will live in a directory called __pycache__/, which is placed in the directory containing the module that provided the imported code
- Inside __pycache__/, you'll find a .pyc file named after your original module plus the interpreter's name and version
- The name of the .pyc file will also include the optimization level used to compile the code

Testing Your Code With Assertions

- Testing is another field in the development process where assertions are useful
  - Testing boils down to comparing an observed value with an expected one to check if they're equal or not
  - This kind of check perfectly fits into assertions
- Assertions must check for conditions that should typically be true, unless you have a bug in your code
  - This idea is another important concept behind testing
- The pytest third-party library is a popular testing framework in Python
  - At its core, you'll find the assert statement, which you can use to write most of your test cases in pytest
- There are a couple of remarkable advantages behind this choice:
  - The assert statement allows pytest to lower the entry barrier and somewhat flatten the learning curve because its users can take advantage of Python syntax that they already know
  - The users of pytest don't need to import anything from the library to start writing test cases. They only need to start importing things if their test cases get complicated, demanding more advanced features
  - These advantages make working with pytest a pleasant experience for beginners and people coming from other testing frameworks with custom APIs
- For example, the standard-library unittest module provides an API consisting of a list of .assert*() methods that work pretty much like assert statements
  - This kind of API can be difficult to learn and memorize for developers starting with the framework

```python
# test_samples.py

def test_sum():
    assert sum([1, 2, 3]) == 6

def test_len():
    assert len([1, 2, 3]) > 0

def test_reversed():
    assert list(reversed([1, 2, 3])) == [3, 2, 1]

def test_membership():
    assert 3 in [1, 2, 3]

def test_isinstance():
    assert isinstance([1, 2, 3], list)

def test_all():
    assert all([True, True, True])

def test_any():
    assert any([False, True, False])

def test_always_fail():
    assert pow(10, 2) == 42
```

1. You can use pytest to run all the test case examples above.
2. First, you need to install the library by issuing the
   python -m pip install pytest
   command.
3. Then you can execute
   pytest test_samples.py
   from the command-line.

Windows下面必须用管理员权限安装。如果不是管理员权限，请删除后再重新安装。或者用 python –m pytest 代替 pytest.

```
(base) ➜  2023 秋季 pytest test_samples.py
============================= test session starts ==============================
platform darwin -- Python 3.9.13, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/fancheng/Library/CloudStorage/OneDrive-Personal/CS124计算导论/2023 秋季
plugins: anyio-3.5.0
collected 8 items

test_samples.py .......F                                                 [100%]

================================== FAILURES ===================================
_____________________________ test_always_fail ______________________________

    def test_always_fail():
>       assert pow(10, 2) == 42
E       assert 100 == 42
E        +  where 100 = pow(10, 2)

test_samples.py:25: AssertionError
=========================== short test summary info ============================
FAILED test_samples.py::test_always_fail - assert 100 == 42
========================= 1 failed, 7 passed in 0.07s ==========================
```

Understanding Common Pitfalls of assert

- Even though assertions are such a great and useful tool, they have some downsides
  - Like any other tool, assertions can be misused.
  - You've learned that you should use assertions mainly for debugging and testing code during development
  - In contrast, you shouldn't rely on assertions to provide functionality in production code, which is one of the main drivers of pitfalls with assertions
- In particular, you may run into pitfalls if you use assertions for:
  - Processing and validating data
  - Handling errors
  - Running operations with side effects
- Another common source of issues with assertions is that keeping them enabled in production can negatively impact your code's performance
- Finally, Python has assertions enabled by default, which can confuse developers coming from other languages