

(1) PYTHON BUILT-INS

Python built-in data types, functions and etc.

1.1 Built-in functions

abs: `__abs__()`, absolute value.
all(iterable): returns `True` if all elements of the iterable are true, or if the iterable is empty.
any(iterable): return `True` if exists true element in the iterable, nor the iterable is empty.
bin, hex, oct (int) -> str: $\text{int} \rightarrow \text{string}$, with `'0b'/'0o'` etc. at the beginning.
chr(int) -> str: converts ascii to a char. Reverses `ord()`.
ord(str) -> int: converts a char to ascii. Reverses `chr()`.
divmod(a, b) := (a // b, a % b)
eval: evaluate the expression.
globals(), locals(): radioactive.
hash(immutable) -> int return the hash of immutable objects.
id(object): unique and constant identity which create for any object.
max, min(iterable, key: function=None) or (*args, key=None) returns the max/min value after applying key, if exists.
sorted(iterable, key=None, reverse=False) -> list: return a new sorted list from the terms in iterable.

1.2 Built-in Data Structures

1.2.1 list

complexity

copy, pop, insert, delete, iteration, slicing: $O(n)$

append, pop last, get, set, len: $O(1)$

sorting: avg. $O(n \log n)$, worst case $O(n^2)$

methods

append(any) -> None, extend(b: iterable) :=
`a[len(a):] = b := a += b, insert(i, object) ->`
`None, pop(int) -> any, count(x) -> int, copy() ->`
`list, reverse() -> None`

(Note that `-> None` mostly means in-place)

1.2.2 dict

complexity, avg.

copy, iteration: $O(n)$

`k in d, get, set, del:` $O(1)$

(For more see 4.4. hash table)

methods

`__init__(iterable [opt.], **kwargs), keys(), values()`
`-> iterable, items() -> iterable[tuple]` (keys and values combined), **`clear() -> None, copy() -> dict,`**
`__init__(iterable[iterable], **kwargs)`

1.2.3 set

methods: `pop, add, remove(elem), pop()` removes random value, `clear()`.

complexity: in, pop, add, remove: avg. $O(1)$, worst $O(n)$.

(For more see 4.4. hash table)

1.3 special methods

assuming `__{name}__` *syx*.

init: initialize, **repr:** `repr(obj)` / REPL representation of the object, **str:** `str(obj)` value, **call:** calling, **getattr, setattr, delattr:** `x.obj, x.obj = x, del obj` respectively, **len:** `len(obj)`, **contains(x):** `x in obj`, **getitem, setitem, delitem,** missing: `obj[key], obj[key] = val, del obj[key], obj[non-existent-key]` respectively, **add, sub, mul, truediv, floordiv, mod, pow (y):** `x + y, x - y, obj * y, x / y, x // y, obj % y` respectively, **iadd, isub (y) etc.:** `obj + y, obj - y` etc. (in-place), **neg, pos, abs, int:** `-x, +x, abs(x), int(x)` respectively, **eq, ne, lt, le, gt, ge, bool (y):** `x == y, x != y, x < y, x <= y, x > y, x >= y`, if `x: [...]` respectively, **hash:** `hash(obj)`.

Note that object's default behaviour is to use `id(obj)` e.g. for **eq, hash** etc.

The most important methods are bolded.

(2) MATHS

2.1 General

Euclid's algo.: $\text{gcd}(a, b) = \text{gcd}(a, b \bmod a)$, $\text{gcd}(a, 0) = a$.

Fermat's Little Theorem: p is prime $\implies \forall a \in [2, \dots, p-1] : a^{p-1} \bmod p = 1$. Ferm_N denoted to be the set of all $a^{N-1} \bmod N \neq 1$ ("fermat-witness").

Prime Witnesses Groups: let n be a number, $\text{Gcd}_n := \{1 < a < N \mid \text{gcd}(N, a) > 1\}$, $\text{Fact}_n = \{a \in \mathbb{N} : a \mid n\}$, then $\text{Fact}_n \subseteq \text{Gcd}_n \subseteq \text{Ferm}_n$.

Miller-Rabin algo.: $|\text{Ferm}_N| \geq \frac{N}{2}$ for every N composite (except for carmichael numbers, which are rare)

Pascal's rule: $\binom{b}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

Hashing to m -sized int.: `hash(x) % m`

A mod rule (for Diffie-Hellman): $(g^a \bmod p)^b = (g^{ab} \bmod p)$

2.2 Master Theorem

let $f: \mathbb{R} \rightarrow \mathbb{R}$ be a function, and let $a \leq 1, b > 1$ be constants, assuming $T: \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$, $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, then:

$$1. \exists \epsilon > 0. f(n) = O(n^{\log_b a - \epsilon})$$

$$\implies T(n) = \Theta(n^{\log_b a})$$

$$2. f(n) = \Theta(n^{\log_b a})$$

$$\implies T(n) = \Theta(n^{\log_b a} \cdot \log n)$$

$$3. \exists \epsilon > 0. f(n) = \Omega(n^{\log_b a + \epsilon}) \wedge$$

$$\exists c > 1, n_0 \geq 0. \forall n \geq n_0. a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$$

$$\implies T(n) = \Theta(f(n))$$

Note that $\frac{n}{b}$ could be $\lfloor \frac{n}{b} \rfloor$ nor $\lceil \frac{n}{b} \rceil$

2.3 Rules for Sums of Series

$$\sum_{i=1}^n i = \frac{1}{2}n(n+1) = \Theta(n^2) \quad (1)$$

$$\sum_{i=0}^n x^i = \frac{x^{n+1}-1}{x-1} = \Theta(x^n) \quad (x \neq 1) \quad (2)$$

$$\sum_{i=0}^{\infty} x^i = (1-x)^{-1} = \Theta(1) \quad (0 < x < 1) \quad (3)$$

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3) \quad (4)$$

$$\sum_{i=1}^n \frac{1}{i} = \Theta(\log n) \quad (5)$$

$$\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4} = \Theta(n^4) \quad (6)$$

$$\sum_{i=1}^n \log i = \Theta(n \log n) \quad (7)$$

$$\sum_{i=1}^n (ca_i + b_i) = c \sum_{i=1}^n a_i + \sum_{i=1}^n b_i \quad (8)$$

$$\sum_{i=1}^n \Theta(f(i)) = \Theta\left(\sum_{i=1}^n f(i)\right) \quad (9)$$

When their names are (1) Arithmetic, (2, 3) Geometric, (4) Square and (5) Hermonic series.

2.4 Asymptotic Barriers

2.4.1 definition

- $f(n) = O(g(n))$ iff $\exists c, n_0 \geq 0. \forall n \geq n_0. f(n) \leq cg(n)$
- $f(n) = \Omega(g(n))$ iff $\exists c, n_0 \geq 0. \forall n \geq n_0. f(n) \geq cg(n)$
- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$

(3) ALGORITHMS

3.1 Sorting & Searching Algo.

3.1.1 Merge Sort

complexity: $\Theta(n \log n)$ (Assuming implementation without slicing)

3.1.2 Quicksort

complexity: best, avg.: $O(n \log n)$, worst: $O(n^2)$. Same as merge sort, with random pivot.

3.1.3 Selection sort

For i in $[0, n]$, find the minimum of $lst[i:]$ and position it at the begining. Always $O(n^2)$. Bad. No real usecase.

3.2 Binary search

When an ordered list is given, find the middle, then repeat for the right/left side of it (larger/smaller). Takes $O(\log n)$

2.4.2 Hierarchy

- | | |
|---|---|
| 1. $\Theta(1)$ (constant) | 8. $\Theta(n^2)$ (quadratic) |
| 2. $\Theta(\log \log n)$ | 9. $\Theta(n^2 \log n)$ |
| 3. $\Theta(\log_a n)$ [$\forall a \geq 2$]
(logarithmic) | 10. $\Theta(2^n)$ (exponential) |
| 4. $\Theta(\log^a n)$ [$\forall a > 1$]
(poly logarithmic) | 11. $\Theta(3^n)$ [etc.] (exponential) |
| 5. $\Theta(\sqrt{n})$ (square root) | 12. $\Theta(n!)$ (factorial) |
| 6. $\Theta(n)$ (linear) | 13. $\Theta(n^n)$ ← your code
(pure bad) |
| 7. $\Theta(n \log n)$ | |

2.4.3 Other rules

$f_1 = O(g_1) \wedge f_2 = O(g_2) \implies f_1 + f_2 = O(\max(g_1, g_2))$
 $\forall a_0, a_1, \dots, a_k \in \mathbb{R}_+, k \geq 1. f(n) = a_0 n^0 + \dots + a_k n^k = \Theta(n^k)$

2.5 Binary Operations

2.5.1 Complexity

Let $n, m, t \in \mathbb{N}$ be natural numbers, and a, b, c their bit size:

Multiplication, Division: $\approx O(ab)$

Addition/Subtraction: $\Theta(\max\{a, b\})$

Floor division by 2: $(n // 2) = O(a)$

Integer Exponent: $(a ** b) = O(\log b)$ multiplications

Modular Exponent: $(n ** m \% t) = O(c^3)$ (assuming $a = b = c$).

2.5.2 Other bases

For number N in base b represented by $a_k a_{k-1} \dots a_1 a_0$, N would be $N = a_k b^k + a_{k-1} b^{k-1} + \dots + a_2 b + a_1$. Hence, $b^{k-1} \leq N \leq b^k - 1$, and $k = \lfloor \log_b N \rfloor + 1$. From this, for a number with d digits in base b , will take at most $\lfloor d \log_c b \rfloor$ digits in base c (assuming $b, c, c > 1$).

3.3 Number Theory Algo.

3.3.1 Modular exponentiation & Interating Square

Complexity: $O(n^3)$, suppose a, b, c are n -bit long.

Interating Square algo.: same as the code below, but without the `%c`.

```
1 def modpow(a, b, c):
2     result = 1
3     while b > 0:
4         if b % 2 == 1: result = (result * a) % c
5         a = a*a % c; b = b//2
6     return result # = a**b%c = pow(a, b, c)
```

3.3.2 pseudo-primes

Returns if N is prime with probability of $1 - 0.5^a$, by fermat's little theorem. $O(n^3)$. (code in the next page)

```

1 def is_prime(N, tests):
2     for i in range(tests):
3         a = random.randint(2, N - 1)
4         if pow(a, N - 1, N) != 1: # a in Ferm_n
5             return False
6         return True

```

3.3.3 GCD (Euclid Algo.)

Complexity: $\approx 2 \log b$ iterations which is $O(\log b)$ (WOLOG $b \geq a$).

```

1 def gcd(a, b):
2     if a < b: a, b = b, a # switch
3     while b > 0:
4         a, b = b, a % b
5     return a

```

3.4 Other

3.4.1 Diffie-Hellman Protocol

Let p be a large prime, and let $1 < g < p - 1$ be a random integer. The algo.:

1. $f(x) = g^x \bmod p$ is a public key
2. Person A chooses number a
3. Person B chooses number B
4. A computes $f(a) = g^a \bmod p$
5. B computes $f(b) = g^b \bmod p$
6. Each one sends the computed number to each other
7. Compute $f(a)^b = f(b)^a$ (according to some random theorem in the math section)

(4) DATA STRUCTURES

4.1 Linked List

Description: Each node stores its value, and the next node location, `None` in case it's the last one. The linked list class stores the head of the list, and the size of the list.

Operation	Linked list	built-in list
insertion after a given element	$O(1)$	$O(n)$
insertion in a given index	$O(n)$	
get / modify n^{th} elements	$O(n)$	$O(1)$
Delete given prev. element	$O(1)$	$O(n)$
Delete by given index	$O(n)$	

4.2 Doubly-Linked List

Each node saves the prev. and next node, and the D.L.L. class saves both the tail (last element) and the head (first element). This method allow us to implement the **rotate** method: given $0 \leq k < m$, the i^{th} node of the list will change place and become the $(i + k) \bmod n$ node (e.g. for $k = 2$, $0 \rightarrow 2$, $-1 \rightarrow 1$ etc.; $+k$ means "right" rotation, $-k$ "left").

4.3 Binary tree

Each node contains information about the next node in the right and left subtrees, and its value. The binary tree class contains info. about the root and the size. Assuming the tree is balanced (deepest in $\log n$), then insert, lookup and minimum takes $\log n$ time.

4.4 Hash table

For m -sized hash table the hash function would be `hash(n) % m`. For each hash value assigned a list, which contains all of the elements with the same hash value. On avg. $\alpha = \frac{m}{n}$ (α called the **load factor**). The `dict` keys and `set` build-in classes uses n -sized hash table, means $O(\frac{n}{n}) = O(1)$ on avg., worst case $O(n)$ (happens when every key is hashed to the same value).

4.5 Generators

4.5.1 Creation

```

1 >>> def gen(*arg, **kwargs): yield val # opt. 1
2 >>> print(type(gen), type(gen()))
3 <class 'function'> <class 'generator'>
4 >>> gen = (val for val in iterable) # opt. 2
5 >>> type(gen)
6 <class 'generator'>

```

4.5.2 Usage

To get the next item, use the `next(gen)` function. We'll get a `StopIteration` at the end of finite generators. e.g.:

```

1 >>> def gen(): yield 1; yield 2; yield 3
2 >>> g = gen()
3 >>> print(next(g), next(g), next(g))
4 1 2 3
5 >>> next(g)
6 Traceback [...]: StopIteration

```

4.5.3 Notes

A given generator has "finite delay" iff the time that takes to generate each item is finite. Generators can be used recursively.

4.6 float

D.A.F.U.K. <an image of a cat>

Saving data: For 64-bit float:

sign[1 bit] + exponent[11 bits] + fraction[52 bits]

Compute: $(-1)^{\text{sign}} \cdot 2^{\text{exponent} - 2023} \cdot (1 + \text{fraction})$

Domain: $0 \leq \text{exponent} \leq 2047$, $0 \leq \text{fraction} \leq \sum_{i=1}^{52} 2^{-i} = 1 - 2^{-52}$

4.7 String Representations

ASCII: 7-bit fixed-length, 00 to 1F / 0-31: nulls, 30 to 39 / 48-57: 0-9, 41-5A / 65-90: A-Z, 61-7A / 98-122: a - b (including).

Unicode: variable length code. Hebrew is between 1488-1514 ($22 + 5 = 27$). The ascii code above works for unicode too.

5.1 Definitions for codes

let $C: \{0,1\}^\Sigma$ be a code;

Universal: $\forall x. |C(x)| < |x|$ when $|x|$ is the raw (binary) length.

There isn't a universal lossless compression scheme.

Codewords: x is a codeword iff $x \in \text{Im}(C)$

Variable-length: $\nexists n. \forall x \in \text{Im}(C). |x| = n$

Prefix-Free: $\forall \tau, \gamma \in \Sigma. \tau \neq \gamma \implies C(\tau)$ isn't a prefix of $C(\gamma)$.

Uniquely-decodable: $\exists C^{-1}: \{0,1\}^n \rightarrow \Sigma^n$ such as $\forall x_0, \dots, x_n \in \text{dom}(C^{-1}). C(C_0^{-1}) = x_0, \dots, C(C_n^{-1}) = x_n$.

5.2 Compression codes

5.2.1 Huffman

Create a Huffman tree from a given corpus:

1. Create a priority queue formatted characters : int
2. Extract 2 minimums
3. Create a tree out of them
4. Add the tree to the queue
5. Goto 2, until you get one tree

Then, each way (left/right) to go to a given character, is decoded to 0/1.

5.3 Lempel-Ziv

Saves repetitions in format of $[m, k]$ when m is the offset backwards, and k is the length of the repetition, e.g.:

```
1 >>> LZW_compress("abcabcabc")
2 ["a", "b", "c", [3, 6]]
```

We denote $1 \leq m \leq W \wedge 1 \leq k \leq L$, when by default $W = 2095 = 2^{12} - 1, L = 31 = 2^5 - 1$ (takes 12, 5 bits respectively).

While converting the above intermediate list to binary, we add 0 before each single ASCII character, and 1 before each $[m, k]$ entry. Hence, assuming ASCII encoding is being used, the minimal length word compressing is $12 + 5 + 1$ bits ($\log W + \log L + 1$ for the prefix), when a given character takes 7 (ASCII) + 1 (prefix) bits. In general, **we'll compress when:**

$$8k > 1 + |W| + |L| + 1, (|X| = \lfloor \log_2 X \rfloor + 1)$$

This following function is being used to find the maximum match within a T text, when p is the current index:

```
1 def maxmatch(T, p, W=2**12-1, L=2**5-1):
2   n = len(T); m = 0; k = 0;
3   for offset in range(1, min(p, W) + 1):
4       match_len, j = 0, p - offset
5       while match_len < min(n - p, L) and \
6             T[j+match_len] == T[p + match_len]:
7           match_len += 1
8       if match_len > k:
9           k, m = match_len, offset
10  return m, k
```

I may have some mistakes

Extended Intro. To Computer Science – Shit Cheat Sheet

Shahar Perets ~ 2024

Made using free software, but not by Stallman's definitions

contact me: sheave.lariat-0h@icloud.com & u/Sh_Pe