| Data Type | iterable | Mutable | indexable | hashable |
|---|---|---|---|---|
| **int** | No | No | No | Yes |
| **float** | No | No | No | Yes |
| **str** | Yes | No | Yes | Yes |
| **tuple** | Yes | No | Yes | Yes |
| **list** | Yes | Yes | Yes | No |
| **set** | Yes | Yes | No | No |
| **dict** | Yes | Yes | Yes | No |

**mutable** (Can change), **indexable** (support get element by a[i]), **hashable** (keep id until lifetime end)
**bool false**: `False`, `None`, `0`, `0.0`, `0j`, "" (empty string), [ ] (empty list) {} (empty dict), `range(0)`, `set()` (empty set)

## Namespace:

> *Built-in*: built-in names, `print`, `int`, `NameError`
> *Global*: global variables
> *Enclosed*: for variables in inside function (wrapped in a function)
> *Local*: For variables in functions
> Each imported module has its own namespace, parallel with local

## Function:

> **`import math`**: import whole math class, when use objects need to type `math.sin()`
> **`from math import sin`**: import sin object, when use only type `sin()`
> **Positional argument**: number and order of the argument is important
> **Keyword argument:** order is not important **`func(x=2, z='123', y=1234)`**
> **Default/optional argument:** can be omitted, has default value
> **Pure function**: function without side-effect (I/O task); only mapping; output depend only on input (can not use global variables)
> **Generator Function**: `return` is changed to **`yield`**. **`yield`** will pause the function, keep the state and resume when the next calling (the value of variable will keep)
> **Inner function:** can read global and outer function variable. Can modify global variable by **`global x`**, can modify **nearest enclosing namespace (outer function)** by **`nonlocal x`**
> Function can read global variable directly, modify global variable by adding `global x`

**Access the global variable:**

> In a function, global variable with `global` can: modify, both mutable and immutable; read
> Global variable without `global` can: modify mutable only by `append` or `sort`, etc.; read

**Pass by assignment**: similar with pointer pass. When a mutable is passed, it will modify the original. For immutable variable, it will create a new object

## Loop and recursion

**`while(x<10)`:** when x<10, keep running. Once x>=10, stop
**Recursive**: Calling itself, solve smaller problem (divide & conquer), more running time for function calling. Often use DP instead of recursive
**Iterative**: for or while loop, faster

## String:

> `'b' in 'banana'` -> True
> `len('hi')` -> 2
> `chr(123)` -> '{' (Unicode to character)
> `ord('{')` -> 123 (Character to ASCII)
> `'string'[0::1]` -> First number: start (inclusive), Second number: end (exclusive), Third number: interval(skip no. of character-1)
> `'IT5001'[0::3]` -> I0 (Start from 'I', skip two)
> `'IT5001'[-1]` -> 1 (last one)
> `'IT5001'[1:3:1]` -> T50 (Start from the second, end at the third, no skip)
> `'I' not in 'IT5001'` -> False (Case sensitive)
> 'abcde'[::-1] -> 'edcba' (reverse)

## List:

> Mutable, can modify the element. Dynamic arrays.
> Can contain one or more types in a list, defined using [ ]
> Can be sorted. **`sort()`** returns a sorted list, **`sorted()`** modify the origin to sorted.
> Can be reversed. **`reverse()`**
> **`a[i]`**, return i-th element of a. **`a[i:j]`**, returns elements i up to j-1. **`len(a)`**, **`min(a)`**, **`max(a)`**. **`x in a`** return True if x is a part of a. **`a+b`**, concatenates a and b. **`n*a`**, creates n copies of sequence a. **Also, for tuple**
> append: **`a.append()`** add an element in the end of list.
> concatenate: **`a+b`**, join two or more lists
> append is same as std::vector, pre-allocate space, fast; concat is slow.
> cannot delete iteratively, since the **`next()`** will be also deleted
> 当创建一个 list 并赋值给一个变量时，该变量实际上只是一个指向对象的引用。这意味着，当在函数中修改一个可变对象（list, dict,）时，你实际上是修改了该对象的内容，而不是引用。也就是说，函数内可以修改函数外的 list。
> **`insert(), pop(), remove(), in, index()`** with O(1), others O(n)

**List Comprehension:** `list = [i for i in range(1,101)]`
**Generator Expression:**

> `list_gen = (i for i in range(1,101))`
> returns an iterator, generate element in demand.
> requires less memory

## Tuple:

> Immutable, cannot be modified, static array.
> Can contain one or more types in a list, defined using ( )
> With only one element: **`tuple1 = (3,)`**
> List usually stores a large collection of data with the **same type**
> Tuple usually stores a small collections of items with **various types**
> **`list()`:** Change tuple to list. **`tuple()`:** change list to tuple
> **`len()`** has O(1), others has O(n)

## Set:

> unordered, mutable, no duplicate elements
> only `len(a)`, `min(a)`, `max(a)`, `x in a`

```
>>> setA = {1,2,3,4}
>>> setB = {3,4,5,6}
>>> setA | setB #Union
{1,2,3,4,5,6}
>>> setA & setB #Intersection
{3,4}
>>> setA - setB #A-B
{1,2}
>>> setA ^ setB #(A|B)-A&B (Symmetric Difference)
{1,2,5,6}
```

> **`add()`,** add single element. **`update()`,** add multiple elements.
> **`delete(), discard()`** remove element, delete will throw error if element is not exist.
> **`pop()`** delete and return an element
> **`clear()`** delete all elements
> **`set(),list()`** for list and set ONLY
> 所有操作都是 $O(1)$,

## Dictionary:

> search key in the dict: `dict.get("apples")` or `dict["apples"]`, each key has a correspondent value.
> 使用 dict.get()时，如果键不存在会返回 None。使用 dict[key]时会返回 KeyError
> each pair has a key(left) and a value(right)
> can store any type
> delete: `dict.pop("apples")` or `del dict["apples"]`
> `clear()`: clear all. `copy()`: make a copy. `keys()`:return all keys.
> `values()`: return all values. `items()`: return all keys + values
> 不能修改正在迭代的容器的大小或结构，但可以改现有键的值
> 如果在插入键值对时,键已经存在,那么此键对应的值会被覆写为新值
> **`dict[key].append()`** is valid
> **`dict.keys(), values(), items()`** 是 $O(n)$, 其余操作都是 O(1)

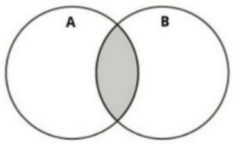
## Lambda:

```
>>> (lambda x:x)(10) # Identity function
10
>>> (lambda x: 'abc')(5) # Constant function
'abc'
>>> (lambda x,y,z: x+y+z)(4,5,9) # Multiple arguments
18
>>> def func_a(n)
        return lambda x:x+n
>>> f1 = func_a(10)
>>> f1(1)
11
```

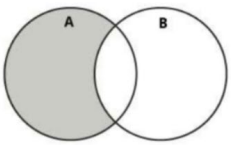
**Variable store a function:**

> `say_hello = greet()`, store the output.
> `say_hello = greet`, store the function
> Their id are identical
> function can store in list, tuple, set, dict
> function can be passed as argument to functions
> function can be returned from function

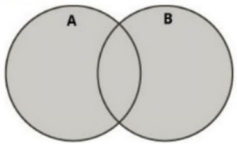
- Intersection



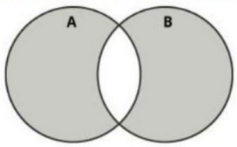

- A – B




- Union




- Symmetric Difference




**Closures:**
> remember the state and the environment
> returns an inner function
> preserve function state across function calls

**Decorators:**
> all decorators are closures
> for decorators, the outer function accepts a function as input arg

## Pastpaper Question

> `int('-12.210')` throws an error, cannot be string
> `['a','b','c','d'][::-1]`即倒序排列，即`['d','c','b','a']`
> `['a','b','c','d'][-1] = ['d']`
> `['a','b','c','d'][1:-1] = ['b','c']`
> `[].append('IT5001')` 不打印任何结果，应该选 none
> `3 in {1,2,{3,4}}` 此 set 包含了一个 set，而 set 中的元素必须是可散列的(hashable)，然而 set 本身是不可散列的，所以此 set 是非法的，同理，tuple 中包含 tuple 也是非法的
> `(lambda x: x (3))(lambda x: x*4)`前面一个 lambda 是带常数的，并且没有附加计算，所以为 3。第一个 lambda 的输出可以看作是第二个 lambda 的输入。即这个表达式可以写为 lambda x: x*4 (3)。所以输出为 12
> `1(2+3)%4` 此表达式中的 1 会被 python 理解为函数名称，所以会抛出错误
> `[1, 2] + (3, 4)` list 和 tuple 不能相加，因为是不同的数据结构，可以将 list 或 tuple 转换成对方的结构后相加。
> `x = [5, 0, 0, 1] += 'IT'` string 是一个可以被迭代的，所以 IT 会被拆分 I 和 T，输出为`[5, 0, 0, 1, 'I', 'T']`
> `[1, 2, 3][4:5] and 'IT5001!'` 其中，`[1, 2, 3][4:5]`会返回一个空列表`[]`，and 会返回第一个逻辑为 false 的值，如果所有制都为 True，则返回最后一个值。空列表被看作是 False，而任何非空的对象都会是 True。所以此表达式会返回空列表`[]`
> `list(filter(bool, [0, 1, 2]))` bool 会测试每个元素的布尔值，然后 filter 会只保留布尔值为 True 的值，所以输出为[1,2]

## Built-in Functions

> round() 舍入到最近的偶数, `round(2.5)=2, round(3.5)=4`
> `input()` 函数总是返回一个字符串类型的值
> `split()` 按照指定的分隔符将字符串拆分成一个子字符串列表，如果没有指定分隔符，那么默认是空白字符，如空格，换行符，制表符。`split('w')`
> `strip()` 移除开头和结尾的指定字符，若未指定，则移除空白字符。还可分为`lstrip()`移除开头的指定字符，`rstrip()`移除末尾指定字符

## 逻辑运算与计算

> 逻辑运算符（如 or）是短路求值的。这意味着如果左边的操作数已经确定了整个表达式的值，那么右边的操作数就不会被评估。
例如 1>1+1 or 3>7-6 or 6>7/0，第二个操作数为 True，那么 python 不会计算之后的操作数，所以不会识别到 ZeroDivisionError：
> 逻辑运算符中，and 的运算顺序要优于 or
> 11&4: 这个表达式使用了按位与运算符 & 来计算两个整数 11 和 4 的二进制按位与结果。 1011 & 0100 = 0000 = 0
> 在多重比较时, 会从左到右进行评估. 例如 False == True == False，其实是 False == True and True == False
> 连续的正负号会按照它们的顺序进行评估。每个`+`操作符不改变数的正负性，而每个`-`操作符会翻转数的正负性。例如`9-+-+-+-9`,第一个`-`将`9`变为`-9`,每个`+`不改变数值,之后的每个`-`翻转数字的正负性，所以最终是`9+9=18`

## Break, continue, pass

> break 用于完全退出循环。continue 用于跳过当前迭代并继续下一个循环。pass 是一个空操作，仅作为占位符

## map() & filter()

> map 和 filter 函数都返回一个可迭代的对象，只能读取一次

## Lambda Function

> Lambda 函数的输入如果在 for loop 中，那么他只会捕获 for loop 的最终值.例如 lambda x: i + x for i in range(3)，此处的 i 只会是 3

## OOP

> super() 函数在 Python 中用于调用父类（或超类）的方法。用法: super().function(arg)
> 当一个类使用了多重继承时 class sub3(sub1, sub2)，那么 super()函数会按照顺序依次调用父类的对应函数，顺序为 sub1, sub1 的父类，sub1 父类的父类...， sub2, sub2 的父类...。若 sub1 和 sub2 共享一个父类，那么 sub1 之后为 sub2
> `__function_name()` is a private member function, use `object._function_name()` to access

**成员函数:**
> 子类会自动继承父类的所有成员函数。如果子类重定义了某个成员函数，那么该成员函数在子类中的版本会覆盖父类中的版本。子类可以通过 super()函数来调用父类的成员函数。

**构造函数 (__init__):**
> 如果子类没有定义自己的构造函数，那么它会自动继承父类的构造函数。如果子类定义了自己的构造函数，那么它需要明确地调用父类的构造函数（如果希望执行父类的构造操作）。这通常通过 super().__init__()来实现。
> 如果子类的构造函数没有调用父类的构造函数，那么父类的构造函数不会自动执行。

**成员变量:**
> 子类不会“继承”父类构造函数中初始化的成员变量的值。但是，如果子类的构造函数调用了父类的构造函数，那么父类的构造函数会执行，从而初始化其成员变量。如果子类和父类具有相同名称的成员变量，并且子类的构造函数没有调用父类的构造函数，那么子类的成员变量会覆盖父类的成员变量。

## Read File

> b for binary format, all mode with b open the file in binary format
> + for reading and writing, all mode with + will open the file with R/W
> r for read only, all mode with r has file ptr at the beginning of the file
> w for write only, overwrite the file if exists, create file otherwise. File ptr at the beginning of the file
> a for appending. File ptr at the end. Create new file if not exist
> `r, rb, r+, rb+, w, wb, w+, wb+, a, ab, a+, ab+`

### Deep Count

```python
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0]) + deepcount(seq[1:])
```

### Deep Map

```python
def deepMap(func,seq):
    if seq == []:
        return seq
    elif type(seq) != list
        return func(seq)
    else:
        return [deepSquare(func,seq[0])] + deepSquare(func,seq[1:])
```

### Change list to tuple deeply

```python
def deep_tuple(s):
    if not isinstance(s, list): return s
    return tuple(deep_tuple(i) for i in s)
```

### Flatten

```python
def flatten(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return [seq]
    else:
        return flatten(seq[0]) + flatten(seq[1:])
```