ADVANCED DATA PROCESSING AND VISUALIZATION TECHNIQUES

Make your analyses smarter

some day in May
Schedule
<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 1st</td>
<td>Holiday</td>
<td>Jun 26th</td>
<td>NoSQL DBs</td>
</tr>
<tr>
<td>May 8th</td>
<td>Intro/Basics</td>
<td>Jul 3rd</td>
<td>ML intro</td>
</tr>
<tr>
<td><strong>May 15th</strong></td>
<td><strong>Pythonics</strong></td>
<td>Jul 10th</td>
<td>Visualization</td>
</tr>
<tr>
<td>May 29nd</td>
<td>Vectorization</td>
<td>Jul 17th</td>
<td>Interactive I</td>
</tr>
<tr>
<td>Jun 5th</td>
<td>Biological DBs</td>
<td>Jul 24th</td>
<td>Interactive II</td>
</tr>
<tr>
<td>Jun 19th</td>
<td>Pandas</td>
<td>Jul 31st</td>
<td>Recap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Aug 7th</td>
<td>Exam</td>
</tr>
</tbody>
</table>

updated: no lecture on May 22nd and June 12th
Schedule Details

- Updated due to the lecture period extension to August 7th
- If there is no lecture the exercise takes place one week later
- Exercises are published on Fridays and discussed Friday the week after
- Last sheet/exercise: Jul 3rd / Jul 10th (maybe adjusted)
- Exam (working date): Aug 7th, to be discussed
Last week, we have covered the basics of the following topics:

- Variables
- Basic data types
- Control structures
- Functions
- Classes
- Modules
Today, we will investigate:

- Objects in Python
- Assignments
- Exceptions
- Iterables & Iterators
- Generators
- Comprehensions & Generator Expressions
- More on Functions
- More on Numbers
- More on Strings
- More on Lists, Tuples, and Dictionaries
We start by saying that everything is an object in python.

```python
def empty_func():
    """This is a doc string"""
    pass
empty_func.__doc__

'This is a doc string'

empty_func.__name__

'emty_func'

(1).__add__(2)

3
Everything is an object

Numbers, strings, lists, tuples, dictionaries, sets, modules, and functions are objects of their corresponding classes.

```python
class MyList(list):
    def length(self):
        return len(self)

my_list = MyList([1, 2, 3])
my_list.length()
```

3
We can assign several values at once:

\[
x, y = 10, 20 \\
y, x \\
20, 10
\]

We can swap values without using a temporary variable:

\[
x, y = y, x \\
x \\
20
\]
We can also use multiple assignments with containers:

```python
my_string = 'Hello, world!'
a, b, c, d, e, f, g, h, i, j, k, l, m = my_string
m = '!
''
```
We have to use a proper number of variables:

```python
my_list = [1, 2, 3, 4, 5]
a, b, c, d = my_list
```

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
(expected 4)

It is customary to assign the values we don’t need to to an 'underscore' variable:

```python
a, b, c, _, _ = my_list
c
3
```
We can assign several values to one variable using the *-operator:

```
my_list = [1, 2, 3, 4, 5]
a, *b = my_list
```

\[
a = 1
\]

\[
b = [2, 3, 4, 5]
\]
Augmented assignments give us a compact way to change a value bound to a variable:

\[
\begin{align*}
\text{x} &= 1 \\
\text{x} &= 1 + 1 \\
\text{x} &= 2 \\
\text{x} &= 2 \times 3 \\
\text{x} &= 6 \times 4 \\
\text{x} &= 24 \\
\text{x} &= 24 \div 6 \\
\text{x} &= 4 \times 5 \\
\text{x} &= 20 \\
\text{x} &= 20 \mod 7 \\
\text{x} &= 6
\end{align*}
\]
We can also use augmented assignment with set operations:

```python
my_set = {1, 2, 3, 4, 5, 6, 7}
my_set ^= {4, 5, 6, 7, 8, 9, 10}
my_set
{1, 2, 3, 8, 9, 10}

my_set -= {1, 2, 3}
my_set
{8, 9, 10}

my_set &= {3, 8, 13}
my_set
{8}

my_set |= {100, 200, 300}
my_set
{8, 100, 300, 200}
```
Conditional expression allows us to use this:

```python
raining = False
weather = 'Bad' if raining else 'Good'
weather
'Good'
```

instead of this:

```python
raining = False
if raining:
    weather = 'Bad'
else:
    weather = 'Good'
weather
'Good'
```
Assignments: Overview

We’ve looked at:

- multiple (tuple, unpacking, iterable unpacking, ...) assignments;
- augmented assignments;
- assignments with ternary operator (also known as conditional expression).
An exception occurs when syntactically correct code results in an error:

```
1 \ 0
```

```
ZeroDivisionError
  Traceback (most recent call last)
<ipython-input-53-b710d87c980c> in <module>()
----> 1 1 / 0

ZeroDivisionError: division by zero
```
Many things can go wrong. We can try to divide by zero, or try to open a file that doesn’t exist, or use a wrong index. In such cases Python program throws an exception and immediately terminates. We can manually throw an exception via `raise`:

```python
raise ValueError('We did it!')
```

```
ValueError
    Traceback (most recent call last)
    <ipython-input-54-29bf2dc22fca> in <module>()
----> 1 raise ValueError('We raised this exception ourselves!')

ValueError: We raised this exception ourselves!
```
To catch an exception, use `try ... except`:

```python
try:
    1 / 0
except:
    print('Oopsie')
Oopsie
```

That said, **NEVER** use an `except` without a specific exception you want to catch!
Python comes with many built-in exception classes. Right now, let’s catch only ZeroDivisionError:

```python
try:
    1 / 0
except ZeroDivisionError as error:
    print(error)

division by zero
```
We can have several `except` clauses:

```python
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Can't divide by zero.")
    except TypeError:
        print("That's not even a number.")
divide(10, 'Hello, World!')
That's not even a number.
```
We can catch different exceptions with one `except` clause:

```python
def divide(a, b):
    try:
        return a / b
    except (ZeroDivisionError, TypeError) as error:
        print(' Seriously?', error)
divide(10, 0)
```

Seriously? division by zero
We can also specify a block of code that should only be run when no exceptions have occurred:

```python
def divide(a, b):
    result = 0
    try:
        result = a / b
    except ZeroDivisionError:
        print('Seriously?')
    else:
        print('Yaaaaaay!')
    return result
```

```
divide(10, 2)
```

```
Yaaaaaay!
5.0
```
Why use `else` instead of putting additional code into the `try` clause? Because that improves readability (we need to investigate fewer lines of code wrapped in the `try` clause) and prevents accidentally catching an exception that is caused by the additional code.
Finally, we can add a block of code that will always be executed as the last task before the `try` statement completes via the `finally` clause.
def squared_divide(a, b):
    result = 0
    try:
        result = a / b
    except ZeroDivisionError as error:
        print(error)
    else:
        result **= 2
    finally:
        print('Cleaning up. ')
    return result

squared_divide(10, 2)

Cleaning up.
25.0
We’ve seen that we can use the `for ... in` loop with strings, lists, tuples, ranges, dictionaries, and sets. Why does it work? What happens behind the scenes?
We could assume that the `for ... in` loop somehow keeps track of the element indices. That assumption wouldn’t work for dictionaries and sets, since we can’t address elements of those containers by index. So, what is really happening?
When we use a `for ... in` loop, it calls the `iter()` function on the container object behind the scenes. The `iter()` function returns an iterator object. This iterator’s `__next()`__ method is then called at each iteration. The `__next()`__ method either returns the next value, or it raises the `StopIteration` exception. If an exception has been raised, the `for ... in` loop catches it and terminates.
items = \{1, 2, 3\}
my_sum = 0
for i in items:
    my_sum += i
print(my_sum)

6

my_sum = 0
items = \{1, 2, 3\}
it = iter(items)
while True:
    try:
        my_sum +=
        next(it)
    except StopIteration:
        break
print(my_sum)

6
That answers one question and immediately raises two other questions:

- what’s an iterator?
- how does \texttt{iter() \textit{create it}}?

Let’s answer the first question: an iterator is an object representing a stream of data. Iterators are required to implement two methods: \texttt{__next()} and \texttt{__iter()}. The \texttt{__iter()} method should return the iterator object itself to allow using an iterator with a \texttt{for...in} loop.
class MyIterator():
    def __init__(self):
        self.current_value = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_value > 2:
            raise StopIteration
        self.current_value += 1
        return self.current_value - 1
for i in MyIterator():
    print(i)

0
1
2
Why would we want to create a custom iterator instead of just putting numbers 0, 1, and 2 in a list? Because we might have situations where the number of values is too large, and we don’t want to load them all in memory at once (iterators are lazy). In other situations the iteration might have complex logic, or we might actually need an infinite number of values.
We can convert a finite iterator into other data types:

<table>
<thead>
<tr>
<th>list(MyIterator())</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 1, 2]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>set(MyIterator())</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0, 1, 2}</td>
</tr>
</tbody>
</table>
We’ve covered iterators, but how does the `iter()` function create them? The function invokes the `__iter__()` method on the passed object. The `__iter__()` method must return an iterator object.

Oh, and by the way, objects that have the `__iter__()` method are iterables.
An **Iterable** is an object that is capable of returning its elements one by one and implements an `__iter__()` method.

An **iterator** is an object that represents a stream of data and implements an `__iter__()` method and an `__next__()` method.

When we use an **iterable** in a `for ... in` loop, an **iterator** is automatically created and is used to produce values until exhaustion.
A generator is a function that returns a generator iterator:

```python
def my_generator():
    i = 0
    while i <= 2:
        yield i
    i += 1

gen_it = my_generator()
for i in gen_it:
    print(i)
```

0
1
2
Python has many built-in classes that are iterators. We will look at the `enumerate` iterator in the following example and then attempt to re-implement it ourselves:

```python
for index, value in enumerate(['Alice', 'Bob', 'Charles']):
    print(str(index) + ' ' + value)
```

```
0 Alice
1 Bob
2 Charles
```
def my_enumerate(iterable):
    index = 0
    for value in iterable:
        yield index, value
        index += 1

for index, value in my_enumerate(['Alice', 'Bob', 'Charles'])�
    print(str(index) + ' ' + value)

0 Alice
1 Bob
2 Charles
Sometimes we want to process two iterables of the same length pairwise (elements with the same indices are related). We might want to do it with `range`:

```python
names = ['Alice', 'Bob', 'Charles']
ages = [23, 21, 71]
for i in range(len(names)):
    print(names[i], 'is', ages[i])
```

Alice is 23
Bob is 21
Charles is 71

**DO NOT DO THIS!**
Alternatively, we might want to do it with `enumerate`:

```
names = ['Alice', 'Bob', 'Charles']
ages = [23, 21, 71]
for i, name in enumerate(names):
    print(name, 'is', ages[i])
```

<table>
<thead>
<tr>
<th>Alice</th>
<th>is 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>is 21</td>
</tr>
<tr>
<td>Charles</td>
<td>is 71</td>
</tr>
</tbody>
</table>

**DO NOT DO THIS EITHER!**
The Pythonic way to solve this problem is via the `zip` function:

```python
names = ['Alice', 'Bob', 'Charles']
ages = [23, 21, 71]
for name, age in zip(names, ages):
    print(name, 'is', age)
```

Alice is 23
Bob is 21
Charles is 71

`zip` wraps two (or more) iterables with a lazy generator that yields tuples containing pairs of next values from each iterable.
Python provides compact syntax to map one list into another list. We have a list of numbers and want to create a list with squares of those numbers. Instead of doing this:

```python
calculated = [2, 3, 4]
squares = []
for number in calculated:
    squares.append(number ** 2)
squares
```

```
[4, 9, 16]
```

we can do this:

```python
squares = [number ** 2 for number in calculated]
squares
```

```
[4, 9, 16]
```
We can filter input elements, keeping only those that satisfy a condition:

```python
even_squares = [n ** 2 for n in numbers if n % 2 == 0]
even_squares
[4, 16]
```
We can loop over nested lists as well:

```python
inner_list_1 = [1, 2, 3]
inner_list_2 = [4, 5, 6]
inner_list_3 = [7, 8, 9]
outer_list = [
    inner_list_1, inner_list_2, inner_list_3
]
squares = [
    n ** 2 for i_l in outer_list for n in i_l
]
squares
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```
A generator expression combines lazy evaluation of generators with the beauty and simplicity of list comprehensions:

```python
count = range(10000)
squares_gen = (n ** 2 for n in count)
type(squares_gen)  # generator
```

```python
for i in range(3):
    print(next(squares_gen))
```

```
0
1
4
```
Set comprehensions allow us to map one set to another in the same fashion:

```python
my_set = {5, 4, 3, 2, 1}
{x ** 2 for x in my_set if x % 2 != 0}
{1, 9, 25}
```
Dictionary comprehensions give us a convenient way to generate dictionaries:

```python
{char: ord(char) for char in 'Hello!'}
```

```python
{
    '!' : 33,
    'H' : 72,
    'e' : 101,
    'l' : 108,
    'o' : 111
}
```
Comprehensions allow us to map one collection of values to another.

We can create lists, sets, and dictionaries through comprehensions.

We can loop over nested collections and perform filtering via comprehensions.

Generator expressions evaluate lazily.
We’ve already seen that functions can accept multiple parameters and can have a return value:

```python
def my_pow(x, power):
    return x ** power

cube_of_four = my_pow(4, 3)
cube_of_four
```

64

Now, we will cover default parameter values, keyword arguments, and arbitrary number of arguments.
We can specify default values for parameters:

```python
def my_pow(x, power=2):
    return x ** power
my_pow(10)
```

100

If we omit the argument for a parameter with a default value as we invoke the function, the default value is automatically used.
Default Parameter Values

Keep in mind that parameters with default values have to appear in the parameter list after the parameters that don’t have defaults.

```python
def my_pow(x=10, power):
    return x ** power

my_pow(2)
```

File ”<ipython-input-84-36e4ccaac92f>”, line 1
def my_pow(x=10, power):
  ^
SyntaxError: non-default argument follows default argument
Keyword arguments allow us to pass arguments in any order when calling a function:

```python
def my_pow(x, power=2):
    return x ** power
my_pow(power=3, x=10)
1000
```
Keyword arguments must be placed after positional arguments at function call.

```python
def my_date(year, month, day):
    print(year, month, day)
my_date(day=15, month=5, 2020)
```

```
File "<ipython-input-88-46ec2682d448>"", line 3
    my_date(day=15, month=5, 2020)
^  
SyntaxError: positional argument follows keyword argument
```
What if we keep the arguments in the same order as parameters?

```
my_date(year=2020, month=5, 15)
```

```
File "<ipython-input-96-7f529f364e5c>"",
    line 1
my_date(year=2020, month=5, 15)
  ^
SyntaxError: positional argument follows keyword argument
```
Keeping positional arguments in front of the keyword arguments works:

```python
my_date(2020, month=5, day=15)
```

2020 5 15
The *-operator allows us to create functions that accept arbitrary number of arguments:

```python
def average(*args):
    return sum(args) / len(args)

average(4, 5, 6, 7, 8)
```

7.0

This function has only one parameter - `args` (this name is used by convention). Its value is a tuple with positional arguments.
We can combine the *-operator with keyword arguments:

```python
def sum_of_powers(*args, power=1):
    return sum([x ** power for x in args])
sum_of_powers(1, 2, 3, 4, power=2)
30

sum_of_powers(5, 10)
15
```

Again, the keyword arguments have to follow the positional arguments.
We can unpack collections and pass their values as arguments to a function:

```python
my_values = [2020, 5, 15]
def my_date(year, month, day):
    print(year, month, day)

my_date(*my_values)
```

2020 5 15
We can do the same with dictionaries and keyword arguments:

```python
my_values = {
    'day': 15,
    'month': 5,
    'year': 2020
}
my_date(**my_values)
```

```
2020 5 15
```
Functions are objects, and we can pass them into other functions as variables. When would we want to use that? Consider the following examples.
We have a list of strings and want to sort them alphabetically. The `sorted` function allows us to do exactly that (keep in mind that it doesn’t sort in place, it returns a new sorted list from the items in an iterable):

```python
strings = ['Bob', 'Charles', 'Alice']
sorted(strings)
```

```
['Alice', 'Bob', 'Charles']
```

That was easy.
We have a list of strings and want to sort them by length. The \texttt{sorted} function accepts a keyword argument \texttt{key=func}, where \texttt{func} is a function of one argument that creates a comparison key for sorting:

```
strings = ['Bob', 'Charles', 'Alice']
sorted(strings, key=len)
```

```
['Bob', 'Alice', 'Charles']
```

That wasn’t difficult either.
We have a list of strings and want to sort them by their second character. We need a function that returns the second character of a given string to use as a key:

```python
strings = ['Bob', 'Charles', 'Alice']
def second_char(string):
    return string[1]
sorted(strings, key=second_char)
['Charles', 'Alice', 'Bob']
```

Things are becoming a bit more complicated. We have to define a function just to perform the sorting we want. There is a better way.
LAMBDA

Lambdas allow us to create a function inline where it’s needed:

```python
sorted(strings, key=lambda x: x[1])

['Charles', 'Alice', 'Bob']
```

A lambda expression creates an anonymous function. The keyword `lambda` is followed by a comma-separated list of parameters, a colon `:`, and an expression (an expression can’t have branches or loops, `return` or `yield` statements). A lambda implicitly returns the value of the expression.
We can use lambdas with functions like `map` and `filter` to respectively map or filter iterables (these functions return an iterator):

```python
list(map(lambda x: x ** 2, [1, 2, 3]))
[1, 4, 9]
```

```python
list(filter(lambda x: x % 2 != 0, [1, 2, 3]))
[1, 3]
```

I would advise to use comprehensions and generator expressions instead of `map` and `filter` whenever possible.
The last point we need to discuss with respect to functions is closures:

```python
def outer_function(outer_param):
    outer_var = 'World!'
    def inner_function():
        print(outer_param, outer_var)
        return inner_function
    inner = outer_function('Hello,')
    inner()

Hello, World!
```
The nested (inner) function can access all the variables of the outer function (that means the outer_param and the outer_var). The outer function defines the inner functions and returns it. The inner function keeps access to the variable of the outer function even after the outer function has finished its execution. This mechanism is used in decorators, which we will discuss later.
Functions can accept positional and keyword arguments.

We can unpack positional and keyword arguments.

Lambdas allow to create an inline function.

We can define a function inside another function.

An inner function can have an access to the variables of the outer function even after the latter has finished its execution.
We have mentioned integers and floating point values during the first lecture. We have also looked into comparisons and basic arithmetic operations. Let’s cover additional numeric data types and additional mathematical operations.
We can create complex numbers using either a literal notation or a `complex` constructor:

<table>
<thead>
<tr>
<th>x = 1+2j</th>
<th>x.real</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.imag</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>complex(1, 2)</td>
<td>(1+2j)</td>
<td></td>
</tr>
</tbody>
</table>
We have mentioned that floats have limited precision:

\[
\begin{align*}
1.1 + 2.2 &= 3.3000000000000003
\end{align*}
\]

Decimals, on the other hand, provide fast exact arithmetic:

```python
from decimal import *
getcontext().prec = 7
dec_1 = Decimal(1.10)
dec_2 = Decimal(2.20)
print(dec_1 + dec_2)
```

\[
3.300000
\]
The fractions module provide support for rational number arithmetic. We can create fractions from integers, floats, decimals, and strings:

```python
from fractions import Fraction
Fraction(1, 2)
Fraction(1, 2)
Fraction(1.5)
Fraction(3, 2)
Fraction('3/7')
Fraction(3, 7)
```
The math module provides additional mathematical functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>math.ceil(5.6)</td>
<td>6</td>
</tr>
<tr>
<td>math.floor(5.6)</td>
<td>5</td>
</tr>
<tr>
<td>math.sqrt(81)</td>
<td>9</td>
</tr>
<tr>
<td>math.cos(0.2)</td>
<td>0.9800665778412416</td>
</tr>
</tbody>
</table>
The statistics module implements mathematical statistics:

```python
import statistics
numbers = [5, 7, 5, 11, 17]
statistics.mean(numbers)
9

statistics.mode(numbers)
5

statistics.median(numbers)
7
```
We can use values of other data types instead of Booleans. Empty collections, zero of any numeric type, and None, are evaluated as False. All the other values are evaluated as True:

```python
number = 11
print('odd' if number % 2 else 'even')
odd
```
We can access several characters at once via slicing:

```python
my_string = 'Hello, World!
my_string[0:5]

'Hello'
```

The start value is set to 0, the stop value is set to 5, and we get all the characters from index 0 to index (5 - 1).

```python
my_string[7:-1]

'World!'
```

The start value is set to 7, the stop value is set to -1, and we get all the characters from index 7 to the very end of the string.
Slicing Strings

We can omit the start and the stop values when we slice from the very beginning or to the very end, respectively:

```
my_string = 'Hello, World!
my_string[:5]

'Hello'
```

If we specify only the stop value, we will get characters of a string from index 0 to index `stop - 1`.

```
my_string[7:]

'World!'
```

If we specify only the start value, we will get characters of a string from index `start` to the end of the string.
We can also specify the step value:

```
my_string[1:10:2]

'el,Wr'
```

Step value can be negative:

```
my_string[:;:-1]

'!dlrow ,olleH'
```
We can format strings in three different ways: the printf-like formatting via the `%`-operator, the `str.format` method, and the f-strings. The printf-like formatting is pretty much obsolete, I do not advise using it:

```
's is approximately %.2f' % ('Pi', 3.14)
'Pi is approximately 3.14'
```
The `str.format` method allows us to substitute values by index or name:

```python
'{}, {}, and {}'.format(
    'first', 'second', 'third'
)

'first, second, and third'
```

We can change the order of the indices:

```python
'{1}, {2}, and {0}'.format(
    'first', 'second', 'third'
)

'second, third, and first'
```
Alternatively, we can use named fields:

```
'{name} is approximately {value}'.format(
    value=3.14, name='Pi'
)

'Pi is approximately 3.14'
```

We can use a dictionary, too:

```
my_dict = {'value': 3.14, 'name': 'Pi'}
'{name} is approximately {value}'.format(
    **my_dict
)

'Pi is approximately 3.14'
```
f-strings allow us to embed Python expressions inside string constants:

```python
n = 3
p = 4

f'\{n\} to the power of \{p\} equals \{n ** p\}\'

'3 to the power of 4 equals 81'
```
Remember to use `dir` and `help`! Strings have many useful methods in Python. `join` and `split` help you to create new strings. `lower`, `upper`, `capitalize`, and `swapcase` allow you to manipulate character case. `lstrip`, `rstrip`, and `strip` remove trailing whitespaces from a string. `startswith`, `endswith`, `isalnum` and others save you from the hassle of using regular expressions.
We can use slicing on tuples and lists in the same way:

\[
(1, 2, 3, 4, 5)[:::-1] \\
(5, 4, 3, 2, 1)
\]

\[
[6, 7, 8, 9, 10][1:-1] \\
[7, 8, 9]
\]
Finally, we will mention ordered dictionaries. Regular dictionaries keep the order of items inserted beginning with Python 3.6, however, that is considered just an implementation detail. If we want to be safe (and get additional features such as checking insertion order when comparing dictionaries), we need to be familiar with OrderedDict.
Ordered Dictionaries

```python
dict_1 = {'a': 1, 'b': 2, 'c': 3}
dict_2 = {'b': 2, 'c': 3, 'a': 1}
dict_1 == dict_2
True

from collections import OrderedDict
o_dict_1 = OrderedDict(
    {'a': 1, 'b': 2, 'c': 3}
)
o_dict_2 = OrderedDict(
    {'b': 2, 'c': 3, 'a': 1}
)
o_dict_1 == o_dict_2
False
```
Further Reading

- https://docs.python.org/3/library/collections.html
- https://docs.python.org/3/library/itertools.html
- https://docs.python.org/3/library/functools.html
- https://docs.python.org/3/library/operator.html