

Hoja de referencia completa sobre Python

Descargar archivo de texto ⁰ [bifurcarme en GitHub](#) [consultar las preguntas frecuentes](#) [cambiar al tema oscuro](#)



```
ToC = {
    '1. Collections': [List, Dictionary, Set, Tuple, Range, Enumerate, Iterator, Generator],
    '2. Types':        [Type, String, Regular_Exp, Format, Numbers, Combinatorics, Datetime],
    '3. Syntax':       [Args, Inline, Import, Decorator, Class, Duck_Types, Enum, Exception],
    '4. System':       [Exit, Print, Input, Command_Line_Arguments, Open, Path, OS_Commands],
    '5. Data':         [JSON, Pickle, CSV, SQLite, Bytes, Struct, Array, Memory_View, Deque],
    '6. Advanced':     [Operator, Match Stmt, Logging, Introspection, Threading, Coroutines],
    '7. Libraries':    [Progress_Bar, Plot, Table, Console_App, GUI, Scraping, Web, Profile],
    '8. Multimedia':   [NumPy, Image, Animation, Audio, Synthesizer, Pygame, Pandas, Plotly]
}
```

Principal

```
if __name__ == '__main__':    # Skips next line if file was imported.
    main()                    # Runs `def main(): ...` function.
```

Lista

```
<list> = [<el_1>, <el_2>, ...] # Creates new list. Also list(<collection>).

<el>   = <list>[index]         # First index is 0. Last -1. Allows assignments.
<list> = <list>[<slice>]        # Also <list>[from_inclusive : to_exclusive : ±step].

<list>.append(<el>)            # Appends element to the end. Also <list> += [<el>].
<list>.extend(<collection>)     # Appends elements to the end. Also <list> += <coll>.

<list>.sort()                  # Sorts elements in ascending order.
<list>.reverse()               # Reverses the list in-place.
<list> = sorted(<collection>)   # Returns new list with sorted elements.
<iter> = reversed(<list>)      # Returns reversed iterator of elements.

<el>   = max(<collection>)      # Returns largest element. Also min(<el_1>, ...).
<num> = sum(<collection>)       # Returns sum of elements. Also math.prod(<coll>).

elementwise_sum = [sum(pair) for pair in zip(list_a, list_b)]
sorted_by_second = sorted(<collection>, key=lambda el: el[1])
sorted_by_both   = sorted(<collection>, key=lambda el: (el[1], el[0]))
flatter_list     = list(itertools.chain.from_iterable(<list>))
```

Para obtener detalles sobre `sort()`, `sorted()`, `min()` y `max()`, consulte [Sortable](#).

El **operador** de módulo tiene la función `itemgetter()` que puede reemplazar las **lambdas** enumeradas.

En este texto se utiliza el término **colección** en lugar de **iterable**. Para conocer la justificación, consulte [Colección](#).

```
<int> = len(<list>)           # Returns number of items. Also works on dict, set and string.
<int> = <list>.count(<el>)     # Returns number of occurrences. Also `if <el> in <coll>: ...`
<int> = <list>.index(<el>)     # Returns index of the first occurrence or raises ValueError.
<el>  = <list>.pop()           # Removes and returns item from the end or at index if passed.
<list>.insert(<int>, <el>)     # Inserts item at index and moves the rest to the right.
<list>.remove(<el>)           # Removes first occurrence of the item or raises ValueError.
<list>.clear()                # Removes all items. Also works on dictionary and set.
```

Diccionario

```
<dict> = {key_1: val_1, key_2: val_2, ...}    # Use `<dict>[key]` to get or set the value.

<view> = <dict>.keys()                  # Collection of keys that reflects changes.
<view> = <dict>.values()                # Collection of values that reflects changes.
<view> = <dict>.items()                 # Coll. of key-value tuples that reflects chgs

value  = <dict>.get(key, default=None)     # Returns default if key is missing.
value  = <dict>.setdefault(key, default=None) # Returns and writes default if key is missing
<dict> = collections.defaultdict(<type>)   # Returns a dict with default value `<type>()`
<dict> = collections.defaultdict(lambda: 1) # Returns a dict with default value 1.

<dict> = dict(<collection>)              # Creates a dict from coll. of key-value pairs
<dict> = dict(zip(keys, values))          # Creates a dict from two collections.
<dict> = dict.fromkeys(keys [, value])    # Creates a dict from collection of keys.

<dict>.update(<dict>)                    # Adds items. Replaces ones with matching keys
value = <dict>.pop(key)                   # Removes item or raises KeyError if missing.
{k for k, v in <dict>.items() if v == value} # Returns set of keys that point to the value.
{k: v for k, v in <dict>.items() if k in keys} # Filters the dictionary by keys.
```

Encimera

```
>>> from collections import Counter
>>> counter = Counter(['blue', 'blue', 'blue', 'red', 'red'])
>>> counter['yellow'] += 1
>>> print(counter.most_common())
[('blue', 3), ('red', 2), ('yellow', 1)]
```

Colocar

```
<set> = {<el_1>, <el_2>, ...}            # Use `set()` for empty set.

<set>.add(<el>)                           # Or: <set> |= {<el>}
<set>.update(<collection> [, ...])         # Or: <set> |= <set>

<set>  = <set>.union(<coll.>)               # Or: <set> | <set>
<set>  = <set>.intersection(<coll.>)        # Or: <set> & <set>
<set>  = <set>.difference(<coll.>)          # Or: <set> - <set>
<set>  = <set>.symmetric_difference(<coll.>) # Or: <set> ^ <set>
<bool> = <set>.issubset(<coll.>)            # Or: <set> <= <set>
<bool> = <set>.issuperset(<coll.>)          # Or: <set> >= <set>

<el> = <set>.pop()                        # Raises KeyError if empty.
<set>.remove(<el>)                        # Raises KeyError if missing.
<set>.discard(<el>)                       # Doesn't raise an error.
```

Conjunto congelado

Es immutable y hashable.

Esto significa que puede usarse como clave en un diccionario o como elemento de un conjunto.

```
<frozenset> = frozenset(<collection>)
```

Tupla

Una tupla es una lista immutable y con capacidad de generar hash.

```
<tuple> = () # Empty tuple.
<tuple> = (<el>,) # Or: <el>,
<tuple> = (<el_1>, <el_2> [, ...]) # Or: <el_1>, <el_2> [, ...]
```

Tupla nombrada

Subclase de una tupla con elementos nombrados.

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', 'x y')
>>> p = Point(1, y=2); p
Point(x=1, y=2)
>>> p[0]
1
>>> p.x
1
>>> getattr(p, 'y')
2
```

Rango

Secuencia immutable y hashable de números enteros.

```
<range> = range(stop) # range(to_exclusive)
<range> = range(start, stop) # range(from_inclusive, to_exclusive)
<range> = range(start, stop, ±step) # range(from_inclusive, to_exclusive, ±step_size)
```

```
>>> [i for i in range(3)]
[0, 1, 2]
```

Enumerar

```
for i, el in enumerate(<coll>, start=0): # Returns next element and its index on each pass.
    ...
```

Iterador

```
<iter> = iter(<collection>) # `iter(<iter>)` returns unmodified iterator.
<iter> = iter(<function>, to_exclusive) # A sequence of return values until 'to_exclusive'.
<el> = next(<iter> [, default]) # Raises StopIteration or returns 'default' on end.
<list> = list(<iter>) # Returns a list of iterator's remaining elements.
```

Herramientas de iteración

```
import itertools as it
```

```

<iter> = it.count(start=0, step=1)           # Returns updated value endlessly. Accepts floats.
<iter> = it.repeat(<el> [, times])           # Returns element endlessly or 'times' times.
<iter> = it.cycle(<collection>)              # Repeats the sequence endlessly.

<iter> = it.chain(<coll>, <coll> [, ...])     # Empties collections in order (figuratively).
<iter> = it.chain.from_iterable(<coll>)       # Empties collections inside a collection in order.

<iter> = it.islice(<coll>, to_exclusive)      # Only returns first 'to_exclusive' elements.
<iter> = it.islice(<coll>, from_inc, ...)     # `to_exclusive, +step_size`. Indices can be None.

```

Generador

Cualquier función que contenga una declaración de rendimiento devuelve un generador.
Los generadores y los iteradores son intercambiables.

```

def count(start, step):
    while True:
        yield start
        start += step

>>> counter = count(10, 2)
>>> next(counter), next(counter), next(counter)
(10, 12, 14)

```

Tipo

Todo es un objeto.
Cada objeto tiene un tipo.
Tipo y clase son sinónimos.

```

<type> = type(<el>)                        # Or: <el>.__class__
<bool> = isinstance(<el>, <type>)          # Or: isinstance(type(<el>), <type>)

>>> type('a'), 'a'.__class__, str
(<class 'str'>, <class 'str'>, <class 'str'>)

```

Algunos tipos no tienen nombres incorporados, por lo que deben importarse:

```

from types import FunctionType, MethodType, LambdaType, GeneratorType, ModuleType

```

Clases base abstractas

Cada clase base abstracta especifica un conjunto de subclases virtuales. Estas clases son reconocidas por `isinstance()` e `issubclass()` como subclases de ABC, aunque en realidad no lo sean. ABC también puede decidir manualmente si una clase específica es o no su subclase virtual, generalmente en función de los métodos que la clase ha implementado. Por ejemplo, Iterable ABC busca el método `iter()`, mientras que Collection ABC busca `iter()`, `contains()` y `len()`.

```

>>> from collections.abc import Iterable, Collection, Sequence
>>> isinstance([1, 2, 3], Iterable)
True

```

	Iterable	Collection	Sequence
list, range, str	✓	✓	✓
dict, set	✓	✓	
iter	✓		

```
>>> from numbers import Number, Complex, Real, Rational, Integral
>>> isinstance(123, Number)
True
```

	Number	Complex	Real	Rational	Integral
int	✓	✓	✓	✓	✓
fractions.Fraction	✓	✓	✓	✓	
float	✓	✓	✓		
complex	✓	✓			
decimal.Decimal	✓				

Cadena

Secuencia inmutable de caracteres.

```
<str> = <str>.strip() # Strips all whitespace characters from both ends
<str> = <str>.strip('<chars>') # Strips passed characters. Also lstrip/rstrip().

<list> = <str>.split() # Splits on one or more whitespace characters.
<list> = <str>.split(sep=None, maxsplit=-1) # Splits on 'sep' str at most 'maxsplit' times.
<list> = <str>.splitlines(keepends=False) # On [\n\r\f\v\x1c-\x1e\x85\u2028\u2029] and \r\n
<str> = <str>.join(<coll_of_strings>) # Joins elements using string as a separator.

<bool> = <sub_str> in <str> # Checks if string contains the substring.
<bool> = <str>.startswith(<sub_str>) # Pass tuple of strings for multiple options.
<int> = <str>.find(<sub_str>) # Returns start index of the first match or -1.
<int> = <str>.index(<sub_str>) # Same, but raises ValueError if there's no match

<str> = <str>.lower() # Changes the case. Also upper/capitalize/title()
<str> = <str>.replace(old, new [, count]) # Replaces 'old' with 'new' at most 'count' times
<str> = <str>.translate(<table>) # Use `str.maketrans(<dict>)` to generate table.

<str> = chr(<int>) # Converts int to Unicode character.
<int> = ord(<str>) # Converts Unicode character to int.
```

Úsalo **'unicodedata.normalize("NFC", <str>)** en cadenas como **'Motörhead'** antes de compararlas con otras cadenas, porque **'ö'** se pueden almacenar como uno o dos caracteres. **'NFC'** convierte dichos caracteres en un solo carácter, mientras que **'NFD'** los convierte en dos.

Métodos de propiedad

```
<bool> = <str>.isdecimal() # Checks for [0-9]. Also [٠-٩] and [᠐-᠙].
<bool> = <str>.isdigit() # Checks for [24321...], and isdecimal().
<bool> = <str>.isnumeric() # Checks for [141234...], [零〇一...], and isdigit().
<bool> = <str>.isalnum() # Checks for [a-zA-Z...], and isnumeric().
<bool> = <str>.isprintable() # Checks for [ !#$%...], and isalnum().
<bool> = <str>.isspace() # Checks for [\t\n\r\f\v\x1c-\x1f\x85\xa0...].
```

Expresiones regulares

Funciones para la coincidencia de expresiones regulares.

```
import re
<str> = re.sub(r'<regex>', new, text, count=0) # Substitutes all occurrences with 'new'.
<list> = re.findall(r'<regex>', text) # Returns all occurrences as strings.
<list> = re.split(r'<regex>', text, maxsplit=0) # Add brackets around regex to keep matches.
<Match> = re.search(r'<regex>', text) # First occurrence of the pattern or None.
<Match> = re.match(r'<regex>', text) # Searches only at the beginning of the text
<iter> = re.finditer(r'<regex>', text) # Returns all occurrences as Match objects.
```

Los literales de cadena sin formato no interpretan secuencias de escape, lo que nos permite utilizar secuencias de escape específicas de expresiones regulares que provocan SyntaxWarning en literales de cadena normales (desde 3.12).

El argumento 'nuevo' de re.sub() puede ser una función que acepta el objeto Match y devuelve una cadena.

El argumento 'flags=re.IGNORECASE' se puede utilizar con todas las funciones.

El argumento 'flags=re.MULTILINE' crea '^' y '\$' coincide con el inicio/final de cada línea.

'flags=re.DOTALL' El argumento '.' también acepta el '\n'.

're.compile(<regex>)' devuelve un objeto Pattern con los métodos sub(), findall(), ...

Objeto coincidente

<str> = <Match>.group()	# Returns the whole match. Also group(0).
<str> = <Match>.group(1)	# Returns part inside the first brackets.
<tuple> = <Match>.groups()	# Returns all bracketed parts.
<int> = <Match>.start()	# Returns start index of the match.
<int> = <Match>.end()	# Returns exclusive end index of the match.

Secuencias especiales

'\d' == '[0-9]'	# Also [0-9...]. Matches a decimal character.
'\w' == '[a-zA-Z0-9_]'	# Also [a-zA-Z0-9_...]. Matches an alphanumeric or _.
'\s' == '[\t\n\r\f\v]'	# Also [\x1c-\x1f...]. Matches a whitespace.

By default, decimal characters and alphanumerics from all alphabets are matched unless 'flags=re.ASCII' is used. It restricts special sequence matches to the first 128 Unicode characters and also prevents '\s' from accepting '\x1c', '\x1d', '\x1e' and '\x1f' (non-printable characters that divide text into files, tables, rows and fields, respectively).

Use a capital letter for negation (all non-ASCII characters will be matched when used in combination with ASCII flag).

Format

<str> = f'{{<el_1>}, {{<el_2>}}'	# Curly brackets can also contain expressions.
<str> = '{}, {}'.format(<el_1>, <el_2>)	# Or: '{0}', {a}'.format(<el_1>, a=<el_2>)
<str> = '%s, %s' % (<el_1>, <el_2>)	# Redundant and inferior C-style formatting.

Example

```
>>> Person = collections.namedtuple('Person', 'name height')
>>> person = Person('Jean-Luc', 187)
>>> f'{person.name} is {person.height / 100} meters tall.'
'Jean-Luc is 1.87 meters tall.'
```

General Options

{<el>:<10}	# '<el>'
{<el>:^10}	# ' ^<el> '
{<el>:>10}	# ' <el> '
{<el>:.<10}	# '<el>.....'
{<el>:0}	# '<el>'

Objects are rendered using 'format(<el>, "<options>")'.

Options can be generated dynamically: f'{{<el>:{{<str/int>}}[...]]'.

Adding '=' to the expression prepends it to the output: f'{{1+1=}}' returns '1+1=2'.

Adding '!r' to the expression converts object to string by calling its repr() method.

Strings

```
{'abcde':10}          # 'abcde'
{'abcde':10.3}         # 'abc'
{'abcde':.3}           # 'abc'
{'abcde'!r:10}         # "'abcde'"
```

Numbers

```
{123456:10}           # '123456'
{123456:10,}          # '123,456'
{123456:10_}          # '123_456'
{123456:+10}          # '+123456'
{123456:+=+10}        # '+123456'
{123456:}             # '123456'
{-123456:}            # '-123456'
```

Floats

```
{1.23456:10.3}        # '1.23'
{1.23456:10.3f}        # '1.235'
{1.23456:10.3e}        # '1.235e+00'
{1.23456:10.3%}        # '123.456%'
```

Comparison of presentation types:

	{<float>}	{<float>:f}	{<float>:e}	{<float>:%}
0.000056789	'5.6789e-05'	'0.000057'	'5.678900e-05'	'0.005679%'
0.00056789	'0.00056789'	'0.000568'	'5.678900e-04'	'0.056789%'
0.0056789	'0.0056789'	'0.005679'	'5.678900e-03'	'0.567890%'
0.056789	'0.056789'	'0.056789'	'5.678900e-02'	'5.678900%'
0.56789	'0.56789'	'0.567890'	'5.678900e-01'	'56.789000%'
5.6789	'5.6789'	'5.678900'	'5.678900e+00'	'567.890000%'
56.789	'56.789'	'56.789000'	'5.678900e+01'	'5678.900000%'

	{<float>:.2}	{<float>:.2f}	{<float>:.2e}	{<float>:.2%}
0.000056789	'5.7e-05'	'0.00'	'5.68e-05'	'0.01%'
0.00056789	'0.00057'	'0.00'	'5.68e-04'	'0.06%'
0.0056789	'0.0057'	'0.01'	'5.68e-03'	'0.57%'
0.056789	'0.057'	'0.06'	'5.68e-02'	'5.68%'
0.56789	'0.57'	'0.57'	'5.68e-01'	'56.79%'
5.6789	'5.7'	'5.68'	'5.68e+00'	'567.89%'
56.789	'5.7e+01'	'56.79'	'5.68e+01'	'5678.90%'

'{<float>:g}' is '{<float>:.6}' with stripped zeros, exponent starting at '1e+06'.

When both rounding up and rounding down are possible, the one that returns result with even last digit is chosen. That makes '{6.5:.0f}' a '6' and '{7.5:.0f}' an '8'.

This rule only effects numbers that can be represented exactly by a float (.5, .25, ...).

Ints

```
{90:c}                # 'Z'. Unicode character with value 90.
{90:b}                # '1011010'. Number 90 in binary.
{90:X}                # '5A'. Number 90 in uppercase hexadecimal.
```

Numbers

```
<int>      = int(<float/str/bool>)          # Or: math.trunc(<float>)
<float>    = float(<int/str/bool>)         # Or: <int/float>e±<int>
<complex>  = complex(real=0, imag=0)       # Or: <int/float> ± <int/float>j
<Fraction> = fractions.Fraction(0, 1)      # Or: Fraction(numerator=0, denominator=1)
<Decimal>  = decimal.Decimal(<str/int>)    # Or: Decimal((sign, digits, exponent))
```

'int(<str>)' and 'float(<str>)' raise ValueError on malformed strings.

Decimal numbers are stored exactly, unlike most floats where **'1.1 + 2.2 != 3.3'**.

Floats can be compared with: **'math.isclose(<float>, <float>)'**.

Precision of decimal operations is set with: **'decimal.getcontext().prec = <int>'**.

Basic Functions

```
<num> = pow(<num>, <num>)                  # Or: <number> ** <number>
<num> = abs(<num>)                         # <float> = abs(<complex>)
<num> = round(<num> [, ±ndigits])          # `round(126, -1) == 130`
```

Math

```
from math import e, pi, inf, nan, isinf, isnan # `<el> == nan` is always False.
from math import sin, cos, tan, asin, acos, atan # Also: degrees, radians.
from math import log, log10, log2              # Log can accept base as second arg.
```

Statistics

```
from statistics import mean, median, variance # Also: stdev, quantiles, groupby.
```

Random

```
from random import random, randint, choice    # Also: shuffle, gauss, triangular, seed.
<float> = random()                            # A float inside [0, 1).
<int>   = randint(from_inc, to_inc)           # An int inside [from_inc, to_inc].
<el>    = choice(<sequence>)                 # Keeps the sequence intact.
```

Bin, Hex

```
<int> = ±0b<bin>                            # Or: ±0x<hex>
<int> = int('±<bin>', 2)                    # Or: int('±<hex>', 16)
<int> = int('±0b<bin>', 0)                  # Or: int('±0x<hex>', 0)
<str> = bin(<int>)                          # Returns '[-]0b<bin>'. Also hex().
```

Bitwise Operators

```
<int> = <int> & <int>                        # And (0b1100 & 0b1010 == 0b1000).
<int> = <int> | <int>                        # Or  (0b1100 | 0b1010 == 0b1110).
<int> = <int> ^ <int>                        # Xor (0b1100 ^ 0b1010 == 0b0110).
<int> = <int> << n_bits                     # Left shift. Use >> for right.
<int> = ~<int>                             # Not. Also -<int> - 1.
```

Combinatorics

```
import itertools as it
```

```
>>> list(it.product([0, 1], repeat=3))
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1),
 (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
```



```
>>> list(it.product('abc', 'abc'))
[('a', 'a'), ('a', 'b'), ('a', 'c'),
 ('b', 'a'), ('b', 'b'), ('b', 'c'),
 ('c', 'a'), ('c', 'b'), ('c', 'c')]
# a b c
# a x x x
# b x x x
# c x x x

>>> list(it.combinations('abc', 2))
[('a', 'b'), ('a', 'c'),
 ('b', 'c')]
# a b c
# a . x x
# b . . x

>>> list(it.combinations_with_replacement('abc', 2))
[('a', 'a'), ('a', 'b'), ('a', 'c'),
 ('b', 'b'), ('b', 'c'),
 ('c', 'c')]
# a b c
# a x x x
# b . x x
# c . . x

>>> list(it.permutations('abc', 2))
[('a', 'b'), ('a', 'c'),
 ('b', 'a'), ('b', 'c'),
 ('c', 'a'), ('c', 'b')]
# a b c
# a . x x
# b x . x
# c x x .
```

Datetime

Provides 'date', 'time', 'datetime' and 'timedelta' classes. All are immutable and hashable.

```
# $ pip3 install python-dateutil
from datetime import date, time, datetime, timedelta, timezone
import zoneinfo, dateutil.tz
```

```
<D> = date(year, month, day) # Only accepts valid dates from 1 to 9999 AD.
<T> = time(hour=0, minute=0, second=0) # Also: `microsecond=0, tzinfo=None, fold=0`.
<DT> = datetime(year, month, day, hour=0) # Also: `minute=0, second=0, microsecond=0, ...`.
<TD> = timedelta(weeks=0, days=0, hours=0) # Also: `minutes=0, seconds=0, microseconds=0`.
```

Aware times and datetimes have defined timezone, while naive don't. If object is naive, it is presumed to be in the system's timezone!

'fold=1' means the second pass in case of time jumping back for one hour.

Timedelta normalizes arguments to \pm days, seconds (< 86400) and microseconds (< 1M). Its str() method returns '[\pm D,]H:MM:SS[...]' and total_seconds() a float of all seconds.

Use '<D/DT>.weekday()' to get the day of the week as an int, with Monday being 0.

Now

```
<D/DTn> = D/DT.today() # Current local date or naive DT. Also DT.now().
<DTa> = DT.now(<tzinfo>) # Aware DT from current time in passed timezone.
```

To extract time use '<DTn>.time()', '<DTa>.time()' or '<DTa>.timetz()'.

Timezone

```
<tzinfo> = timezone.utc # London without daylight saving time (DST).
<tzinfo> = timezone(<timedelta>) # Timezone with fixed offset from UTC.
<tzinfo> = dateutil.tz.tzlocal() # Local timezone with dynamic offset from UTC.
<tzinfo> = zoneinfo.ZoneInfo('<iana_key>') # 'Continent/City_Name' zone with dynamic offset.
<DTa> = <DT>.astimezone([<tzinfo>]) # Converts DT to the passed or local fixed zone.
<Ta/DTa> = <T/DT>.replace(tzinfo=<tzinfo>) # Changes object's timezone without conversion.
```

Timezones returned by tzlocal(), ZoneInfo() and implicit local timezone of naive objects have offsets that vary through time due to DST and historical changes of the base offset.

To get ZoneInfo() to work on Windows run '> pip3 install tzdata'.

Encode

```
<D/T/DT> = D/T/DT.fromisoformat(<str>)      # Object from ISO string. Raises ValueError.
<DT>      = DT.strptime(<str>, '<format>')    # Datetime from str, according to format.
<D/DTn>   = D/DT.fromordinal(<int>)          # D/DT from days since the Gregorian NYE 1.
<DTn>     = DT.fromtimestamp(<float>)         # Local naive DT from seconds since the Epoch.
<DTa>     = DT.fromtimestamp(<float>, <tz>)  # Aware datetime from seconds since the Epoch.
```

ISO strings come in following forms: **'YYYY-MM-DD'**,
'HH:MM:SS.mmmuuu[±HH:MM]', or both separated by an arbitrary character. All
parts following the hours are optional.
Python uses the Unix Epoch: **'1970-01-01 00:00 UTC'**, **'1970-01-01 01:00
CET'**, ...

Decode

```
<str>      = <D/T/DT>.isoformat(sep='T')      # Also `timespec='auto/hours/minutes/seconds/...'`.
<str>      = <D/T/DT>.strftime('<format>')    # Custom string representation of the object.
<int>      = <D/DT>.toordinal()               # Days since Gregorian NYE 1, ignoring time and tz
<float>    = <DTn>.timestamp()                # Seconds since the Epoch, from local naive DT.
<float>    = <DTa>.timestamp()                # Seconds since the Epoch, from aware datetime.
```

Format

```
>>> dt = datetime.strptime('2025-08-14 23:39:00.00 +0200', '%Y-%m-%d %H:%M:%S.%f %z')
>>> dt.strftime("%dth of %B '%y (%a), %I:%M %p %Z")
"14th of August '25 (Thu), 11:39 PM UTC+02:00"
```

'%Z' accepts **'±HH[:MM]'** and returns **'±HHMM'** or empty string if datetime is naive.
'%Z' accepts **'UTC/GMT'** and local timezone's code and returns timezone's name,
'UTC[±HH:MM]' if timezone is nameless, or an empty string if datetime is naive.

Arithmetics

```
<bool>     = <D/T/DTn> > <D/T/DTn>           # Ignores time jumps (fold attribute). Also ==.
<bool>     = <DTa> > <DTa>                   # Ignores time jumps if they share tzinfo object.
<TD>       = <D/DTn> - <D/DTn>               # Ignores jumps. Convert to UTC for actual delta.
<TD>       = <DTa> - <DTa>                   # Ignores jumps if they share tzinfo object.
<D/DT>     = <D/DT> ± <TD>                   # Returned datetime can fall into missing hour.
<TD>       = <TD> * <float>                  # Also `<TD> = abs(<TD>)` , `<TD> = <TD> ± <TD>`.
<float>    = <TD> / <TD>                    # Also `<int>, <TD> = divmod(<TD>, <TD>)`.
```

Arguments

Inside Function Call

```
func(<positional_args>)          # func(0, 0)
func(<keyword_args>)             # func(x=0, y=0)
func(<positional_args>, <keyword_args>) # func(0, y=0)
```

Inside Function Definition

```
def func(<nondefault_args>): ...      # def func(x, y): ...
def func(<default_args>): ...         # def func(x=0, y=0): ...
def func(<nondefault_args>, <default_args>): ... # def func(x, y=0): ...
```

Default values are evaluated when function is first encountered in the scope.
Any mutation of a mutable default value will persist between invocations!

Splat Operator

Inside Function Call

Splat expands a collection into positional arguments, while splatty-splat expands a dictionary into keyword arguments.

```
args = (1, 2)
kwargs = {'x': 3, 'y': 4, 'z': 5}
func(*args, **kwargs)
```

Is the same as:

```
func(1, 2, x=3, y=4, z=5)
```

Inside Function Definition

Splat combines zero or more positional arguments into a tuple, while splatty-splat combines zero or more keyword arguments into a dictionary.

```
def add(*a):
    return sum(a)
```

```
>>> add(1, 2, 3)
6
```

Legal argument combinations:

```
def f(*args): ...           # f(1, 2, 3)
def f(x, *args): ...        # f(1, 2, 3)
def f(*args, z): ...         # f(1, 2, z=3)
```

```
def f(**kwargs): ...        # f(x=1, y=2, z=3)
def f(x, **kwargs): ...      # f(x=1, y=2, z=3) | f(1, y=2, z=3)
```

```
def f(*args, **kwargs): ...  # f(x=1, y=2, z=3) | f(1, y=2, z=3) | f(1, 2, z=3) | f(1, 2, 3)
def f(x, *args, **kwargs): ... # f(x=1, y=2, z=3) | f(1, y=2, z=3) | f(1, 2, z=3) | f(1, 2, 3)
def f(*args, y, **kwargs): ... # f(x=1, y=2, z=3) | f(1, y=2, z=3)
```

```
def f(*, x, y, z): ...       # f(x=1, y=2, z=3)
def f(x, *, y, z): ...       # f(x=1, y=2, z=3) | f(1, y=2, z=3)
def f(x, y, *, z): ...       # f(x=1, y=2, z=3) | f(1, y=2, z=3) | f(1, 2, z=3)
```

Other Uses

```
<list> = [*<coll.> [, ...]]  # Or: list(<collection>) [+ ...]
<tuple> = (*<coll.>, [...])   # Or: tuple(<collection>) [+ ...]
<set> = {*<coll.> [, ...]}    # Or: set(<collection>) [| ...]
<dict> = {**<dict> [, ...]}   # Or: <dict> | ...
```

```
head, *body, tail = <coll.>    # Head or tail can be omitted.
```

Inline

Lambda

```
<func> = lambda: <return_value>           # A single statement function.
<func> = lambda <arg_1>, <arg_2>: <return_value> # Also allows default arguments.
```

Comprehensions

```
<list> = [i+1 for i in range(10)]           # Or: [1, 2, ..., 10]
<iter> = (i for i in range(10) if i > 5)    # Or: iter([6, 7, 8, 9])
<set> = {i+5 for i in range(10)}           # Or: {5, 6, ..., 14}
<dict> = {i: i*2 for i in range(10)}       # Or: {0: 0, 1: 2, ..., 9: 18}
```

```
>>> [l+r for l in 'abc' for r in 'abc']    # Inner loop is on the right side.
['aa', 'ab', 'ac', ..., 'cc']
```

Map, Filter, Reduce

```
from functools import reduce
```

```
<iter> = map(lambda x: x + 1, range(10))    # Or: iter([1, 2, ..., 10])
<iter> = filter(lambda x: x > 5, range(10)) # Or: iter([6, 7, 8, 9])
<obj> = reduce(lambda out, x: out + x, range(10)) # Or: 45
```

Any, All

```
<bool> = any(<collection>)                # Is `bool(<el>)` True for any el?
<bool> = all(<collection>)                 # True for all? Also True if empty.
```

Conditional Expression

```
<obj> = <exp> if <condition> else <exp>     # Only one expression is evaluated.
```

```
>>> [a if a else 'zero' for a in (0, 1, 2, 3)] # `any([0, '', [], None]) == False`
['zero', 1, 2, 3]
```

Named Tuple, Enum, Dataclass

```
from collections import namedtuple
Point = namedtuple('Point', 'x y')          # Creates a tuple's subclass.
point = Point(0, 0)                        # Returns its instance.
```

```
from enum import Enum
Direction = Enum('Direction', 'N E S W')   # Creates an enum.
direction = Direction.N                     # Returns its member.
```

```
from dataclasses import make_dataclass
Player = make_dataclass('Player', ['loc', 'dir']) # Creates a class.
player = Player(point, direction)             # Returns its instance.
```

Imports

Mechanism that makes code in one file available to another file.

```
import <module>           # Imports a built-in or '<module>.py'.
import <package>          # Imports a built-in or '<package>/__init__.py'.
import <package>.<module> # Imports a built-in or '<package>/<module>.py'.
```

Package is a collection of modules, but it can also define its own objects.

On a filesystem this corresponds to a directory of Python files with an optional init script.

Running **'import <package>'** does not automatically provide access to the package's modules unless they are explicitly imported in its init script.

Directory of the file that is passed to python command serves as a root of local imports.

For relative imports use **'from .[...][<pkg/module>[...]] import <obj>'**.

Closure

We have/get a closure in Python when a nested function references a value of its enclosing function and then the enclosing function returns its nested function.

```
def get_multiplier(a):  
    def out(b):  
        return a * b  
    return out
```

```
>>> multiply_by_3 = get_multiplier(3)  
>>> multiply_by_3(10)  
30
```

Any value that is referenced from within multiple nested functions gets shared.

Partial

```
from functools import partial  
<function> = partial(<function> [, <arg_1> [, ...]])
```

```
>>> def multiply(a, b):  
...     return a * b  
>>> multiply_by_3 = partial(multiply, 3)  
>>> multiply_by_3(10)  
30
```

Partial is also useful in cases when a function needs to be passed as an argument because it enables us to set its arguments beforehand.

A few examples being: '`defaultdict(<func>)`', '`iter(<func>, to_exc)`' and dataclass's '`field(default_factory=<func>)`'.

Non-Local

If variable is being assigned to anywhere in the scope, it is regarded as a local variable, unless it is declared as a 'global' or a 'nonlocal'.

```
def get_counter():  
    i = 0  
    def out():  
        nonlocal i  
        i += 1  
        return i  
    return out
```

```
>>> counter = get_counter()  
>>> counter(), counter(), counter()  
(1, 2, 3)
```

Decorator

A decorator takes a function, adds some functionality and returns it.
It can be any **callable**, but is usually implemented as a function that returns a closure.

```
@decorator_name
def function_that_gets_passed_to_decorator():
    ...
```

Debugger Example

Decorator that prints function's name every time the function is called.

```
from functools import wraps

def debug(func):
    @wraps(func)
    def out(*args, **kwargs):
        print(func.__name__)
        return func(*args, **kwargs)
    return out

@debug
def add(x, y):
    return x + y
```

Wraps is a helper decorator that copies the metadata of the passed function (func) to the function it is wrapping (out). Without it, 'add.__name__' would return 'out'.

Cache

Decorator that caches function's return values. All function's arguments must be hashable.

```
from functools import cache

@cache
def fib(n):
    return n if n < 2 else fib(n-2) + fib(n-1)
```

Potential problem with cache is that it can grow indefinitely. To clear stored values run 'fib.cache_clear()', or use '@lru_cache(maxsize=<int>)' decorator instead.

CPython interpreter limits recursion depth to 3000 by default. To increase it run 'sys.setrecursionlimit(<int>)'.

Parametrized Decorator

A decorator that accepts arguments and returns a normal decorator that accepts a function.

```
from functools import wraps

def debug(print_result=False):
    def decorator(func):
        @wraps(func)
        def out(*args, **kwargs):
            result = func(*args, **kwargs)
            print(func.__name__, result if print_result else '')
            return result
        return out
    return decorator

@debug(print_result=True)
def add(x, y):
    return x + y
```

Using only '@debug' to decorate the add() function would not work here, because debug would then receive the add() function as a 'print_result' argument. Decorators can however manually check if the argument they received is a function and act accordingly.

Class

A template for creating user-defined objects.

```
class MyClass:
    def __init__(self, a):
        self.a = a
    def __str__(self):
        return str(self.a)
    def __repr__(self):
        class_name = self.__class__.__name__
        return f'{class_name}({self.a!r})'

    @classmethod
    def get_class_name(cls):
        return cls.__name__
```

```
>>> obj = MyClass(1)
>>> obj.a, str(obj), repr(obj)
(1, '1', 'MyClass(1)')
```

Return value of str() should be readable and of repr() unambiguous.

If only repr() is defined, it will also be used for str().

Methods decorated with '@staticmethod' do not receive 'self' nor 'cls' as their first argument.

Expressions that call the str() method:

```
print(<obj>)
f'{<obj>}'
logging.warning(<obj>)
csv.writer(<file>).writerow([<obj>])
raise Exception(<obj>)
```

Expressions that call the repr() method:

```
print/str/repr([<obj>])
print/str/repr({<obj>: <obj>})
f'{<obj>!r}'
Z = dataclasses.make_dataclass('Z', ['a']); print/str/repr(Z(<obj>))
>>> <obj>
```

Inheritance

```
class Person:
    def __init__(self, name):
        self.name = name

class Employee(Person):
    def __init__(self, name, staff_num):
        super().__init__(name)
        self.staff_num = staff_num
```

Multiple inheritance:

```
class A: pass
class B: pass
class C(A, B): pass
```

MRO determines the order in which parent classes are traversed when searching for a method or an attribute:

```
>>> C.mro()
[<class 'C'>, <class 'A'>, <class 'B'>, <class 'object'>]
```

Type Annotations

They add type hints to variables, arguments and functions ('def f() -> <type>:').

Hints are used by type checkers like [mypy](#), data validation libraries such as [Pydantic](#) and lately also by [Cython](#) compiler. However, they are not enforced by CPython interpreter.

```
from collections import abc
```

```
<name>: <type> [| ...] [= <obj>] # `|` since 3.10.
<name>: list/set/abc.Iterable/abc.Sequence[<type>] [= <obj>] # Since 3.9.
<name>: dict/tuple[<type>, ...] [= <obj>] # Since 3.9.
```

Dataclass

Decorator that uses class variables to generate `init()`, `repr()` and `eq()` special methods.

```
from dataclasses import dataclass, field, make_dataclass

@dataclass(order=False, frozen=False)
class <class_name>:
    <attr_name>: <type>
    <attr_name>: <type> = <default_value>
    <attr_name>: list/dict/set = field(default_factory=list/dict/set)
```

Objects can be made **sortable** with '**order=True**' and immutable with '**frozen=True**'.

For object to be **hashable**, all attributes must be hashable and 'frozen' must be True. Function `field()` is needed because '**<attr_name>: list = []**' would make a list that is shared among all instances. Its '**default_factory**' argument can be any **callable**. For attributes of arbitrary type use '**typing.Any**'.

```
Point = make_dataclass('Point', ['x', 'y'])
Point = make_dataclass('Point', [('x', float), ('y', float)])
Point = make_dataclass('Point', [('x', float, 0), ('y', float, 0)])
```


Property

Pythonic way of implementing getters and setters.

```
class Person:
    @property
    def name(self):
        return ' '.join(self._name)

    @name.setter
    def name(self, value):
        self._name = value.split()
```

```
>>> person = Person()
>>> person.name = '\t Guido  van Rossum \n'
>>> person.name
'Guido van Rossum'
```

Slots

Mechanism that restricts objects to attributes listed in 'slots', reduces their memory footprint.

```
class MyClassWithSlots:
    __slots__ = ['a']
    def __init__(self):
        self.a = 1
```

Copy

```
from copy import copy, deepcopy
<object> = copy/deepcopy(<object>)
```

Duck Types

A duck type is an implicit type that prescribes a set of special methods. Any object that has those methods defined is considered a member of that duck type.

Comparable

If eq() method is not overridden, it returns '**id(self) == id(other)**', which is the same as '**self is other**'.

That means all user-defined objects compare not equal by default.

Only the left side object has eq() method called, unless it returns NotImplemented, in which case the right object is consulted. False is returned if both return NotImplemented.

Ne() automatically works on any object that has eq() defined.

```
class MyComparable:
    def __init__(self, a):
        self.a = a
    def __eq__(self, other):
        if isinstance(other, type(self)):
            return self.a == other.a
        return NotImplemented
```

Hashable

Hashable object needs both `hash()` and `eq()` methods and its hash value should never change.

Hashable objects that compare equal must have the same hash value, meaning default `hash()` that returns `'id(self)'` will not do.

That is why Python automatically makes classes unhashable if you only implement `eq()`.

```
class MyHashable:
    def __init__(self, a):
        self._a = a
    @property
    def a(self):
        return self._a
    def __eq__(self, other):
        if isinstance(other, type(self)):
            return self.a == other.a
        return NotImplemented
    def __hash__(self):
        return hash(self.a)
```

Sortable

With `'total_ordering'` decorator, you only need to provide `eq()` and one of `lt()`, `gt()`, `le()` or `ge()` special methods and the rest will be automatically generated.

Functions `sorted()` and `min()` only require `lt()` method, while `max()` only requires `gt()`. However, it is best to define them all so that confusion doesn't arise in other contexts.

When two lists, strings or dataclasses are compared, their values get compared in order until a pair of unequal values is found. The comparison of this two values is then returned. The shorter sequence is considered smaller in case of all values being equal.

To sort collection of strings in proper alphabetical order pass

`'key=locale.strxfrm'` to `sorted()` after running

`'locale.setlocale(locale.LC_COLLATE, "en_US.UTF-8")'`.

```
from functools import total_ordering
@total_ordering
class MySortable:
    def __init__(self, a):
        self.a = a
    def __eq__(self, other):
        if isinstance(other, type(self)):
            return self.a == other.a
        return NotImplemented
    def __lt__(self, other):
        if isinstance(other, type(self)):
            return self.a < other.a
        return NotImplemented
```

Iterator

Any object that has methods `next()` and `iter()` is an iterator.

`Next()` should return next item or raise `StopIteration` exception.

`Iter()` should return `'self'`, i.e. unmodified object on which it was called.

```
class Counter:
    def __init__(self):
        self.i = 0
    def __next__(self):
        self.i += 1
        return self.i
    def __iter__(self):
        return self
```

```
>>> counter = Counter()
>>> next(counter), next(counter), next(counter)
(1, 2, 3)
```

Python has many different iterator objects:

Sequence iterators returned by the `iter()` function, such as `list_iterator` and `set_iterator`.

Objects returned by the `itertools` module, such as `count`, `repeat` and `cycle`.

Generators returned by the `generator functions` and `generator expressions`.

File objects returned by the `open()` function, etc.

Callable

All functions and classes have a `call()` method, hence are callable.

Use `'callable(<obj>)'` or `'isinstance(<obj>, collections.abc.Callable)'` to check if object is callable.

When this cheatsheet uses `'<function>'` as an argument, it means `'<callable>'`.

```
class Counter:
    def __init__(self):
        self.i = 0
    def __call__(self):
        self.i += 1
        return self.i
```

```
>>> counter = Counter()
>>> counter(), counter(), counter()
(1, 2, 3)
```

Context Manager

With statements only work on objects that have `enter()` and `exit()` special methods.

`Enter()` should lock the resources and optionally return an object.

`Exit()` should release the resources.

Any exception that happens inside the `with` block is passed to the `exit()` method.

The `exit()` method can suppress the exception by returning a `true` value.

```
class MyOpen:
    def __init__(self, filename):
        self.filename = filename
    def __enter__(self):
        self.file = open(self.filename)
        return self.file
    def __exit__(self, exc_type, exception, traceback):
        self.file.close()
```

```
>>> with open('test.txt', 'w') as file:
...     file.write('Hello World!')
>>> with MyOpen('test.txt') as file:
...     print(file.read())
Hello World!
```

Iterable Duck Types

Iterable

Only required method is `iter()`. It should return an iterator of object's items.
`Contains()` automatically works on any object that has `iter()` defined.

```
class MyIterable:
    def __init__(self, a):
        self.a = a
    def __iter__(self):
        return iter(self.a)
    def __contains__(self, el):
        return el in self.a
```

```
>>> obj = MyIterable([1, 2, 3])
>>> [el for el in obj]
[1, 2, 3]
>>> 1 in obj
True
```

Collection

Only required methods are `iter()` and `len()`. `len()` should return the number of items.
This cheatsheet actually means '<iterable>' when it uses '<collection>'.
I chose not to use the name 'iterable' because it sounds scarier and more vague than 'collection'. The main drawback of this decision is that the reader could think a certain function doesn't accept iterators when it does, since iterators are the only built-in objects that are iterable but are not collections.

```
class MyCollection:
    def __init__(self, a):
        self.a = a
    def __iter__(self):
        return iter(self.a)
    def __contains__(self, el):
        return el in self.a
    def __len__(self):
        return len(self.a)
```

Sequence

Only required methods are `getitem()` and `len()`.
`getitem()` should return an item at the passed index or raise `IndexError`.
`iter()` and `contains()` automatically work on any object that has `getitem()` defined.
`Reversed()` automatically works on any object that has `getitem()` and `len()` defined. It returns reversed iterator of object's items.

```
class MySequence:
    def __init__(self, a):
        self.a = a
    def __iter__(self):
        return iter(self.a)
    def __contains__(self, el):
        return el in self.a
    def __len__(self):
        return len(self.a)
    def __getitem__(self, i):
        return self.a[i]
    def __reversed__(self):
        return reversed(self.a)
```

Discrepancies between glossary definitions and abstract base classes:

Python's glossary defines iterable as any object with special methods `iter()` and/or `getitem()` and sequence as any object with `getitem()` and `len()`. It doesn't define collection.

Passing ABC Iterable to `isinstance()` or `issubclass()` only checks whether object/class has special method `iter()`, while ABC Collection checks for `iter()`, `contains()` and `len()`.

ABC Sequence

It's a richer interface than the basic sequence.

Extending it generates `iter()`, `contains()`, `reversed()`, `index()` and `count()`.

Unlike `'abc.Iterable'` and `'abc.Collection'`, it is not a duck type. That is why `'issubclass(MySequence, abc.Sequence)'` would return False even if `MySequence` had all the methods defined. It however recognizes list, tuple, range, str, bytes, bytearray, array, memoryview and deque, since they are registered as Sequence's virtual subclasses.

```
from collections import abc

class MyAbcSequence(abc.Sequence):
    def __init__(self, a):
        self.a = a
    def __len__(self):
        return len(self.a)
    def __getitem__(self, i):
        return self.a[i]
```

Table of required and automatically available special methods:

	Iterable	Collection	Sequence	abc.Sequence
<code>iter()</code>	!	!	✓	✓
<code>contains()</code>	✓	✓	✓	✓
<code>len()</code>		!	!	!
<code>getitem()</code>			!	!
<code>reversed()</code>			✓	✓
<code>index()</code>				✓
<code>count()</code>				✓

Method `iter()` is required for `'isinstance(<obj>, abc.Iterable)'` to return True, however any object with `getitem()` will work with any code expecting an iterable.

MutableSequence, Set, MutableSet, Mapping and MutableMapping ABCs are also extendable. Use `'<abc>.__abstractmethods__'` to get names of required methods.

Enum

Class of named constants called members.

```
from enum import Enum, auto
```

```
class <enum_name>(Enum):
    <member_name> = auto()           # Increment of the last numeric value or 1.
    <member_name> = <value>          # Values don't have to be hashable.
    <member_name> = <el_1>, <el_2>    # Values can be collections (this is a tuple).
```

Methods receive the member they were called on as the 'self' argument.

Accessing a member named after a reserved keyword causes `SyntaxError`.

```

<member> = <enum>.<member_name>           # Returns a member. Raises AttributeError.
<member> = <enum>['<member_name>']         # Returns a member. Raises KeyError.
<member> = <enum>(<value>)                 # Returns a member. Raises ValueError.
<str>     = <member>.name                 # Returns member's name.
<obj>     = <member>.value                # Returns member's value.

<list>    = list(<enum>)                  # Returns enum's members.
<list>    = [a.name for a in <enum>]       # Returns enum's member names.
<list>    = [a.value for a in <enum>]      # Returns enum's member values.

<enum>    = type(<member>)                # Returns member's enum.
<iter>    = itertools.cycle(<enum>)       # Returns endless iterator of members.
<member> = random.choice(list(<enum>))    # Returns a random member.

```

Inline

```

Cutlery = Enum('Cutlery', 'FORK KNIFE SPOON')
Cutlery = Enum('Cutlery', ['FORK', 'KNIFE', 'SPOON'])
Cutlery = Enum('Cutlery', {'FORK': 1, 'KNIFE': 2, 'SPOON': 3})

```

User-defined functions cannot be values, so they must be wrapped:

```

from functools import partial
LogicOp = Enum('LogicOp', {'AND': partial(lambda l, r: l and r),
                           'OR':  partial(lambda l, r: l or r)})

```

Exceptions

```

try:
    <code>
except <exception>:
    <code>

```

Complex Example

```

try:
    <code_1>
except <exception_a>:
    <code_2_a>
except <exception_b>:
    <code_2_b>
else:
    <code_2_c>
finally:
    <code_3>

```

Code inside the **'else'** block will only be executed if **'try'** block had no exceptions.

Code inside the **'finally'** block will always be executed (unless a signal is received).

All variables that are initialized in executed blocks are also visible in all subsequent blocks, as well as outside the try statement (only function block delimits scope).

To catch signals use **'signal.signal(signal_number, <func>)'**.

Catching Exceptions

```

except <exception>: ...
except <exception> as <name>: ...
except (<exception>, [...]): ...
except (<exception>, [...]) as <name>: ...

```

Also catches subclasses of the exception.

Use **'traceback.print_exc()'** to print the full error message to stderr.

Use `'print(<name>)'` to print just the cause of the exception (its arguments).
 Use `'logging.exception(<str>)'` to log the passed message, followed by the full error message of the caught exception. For details see [Logging](#).
 Use `'sys.exc_info()'` to get exception type, object, and traceback of caught exception.

Raising Exceptions

```
raise <exception>
raise <exception>()
raise <exception>(<obj> [, ...])
```

Re-raising caught exception:

```
except <exception> [as <name>]:
    ...
    raise
```

Exception Object

```
arguments = <name>.args
exc_type = <name>.__class__
filename = <name>.__traceback__.tb_frame.f_code.co_filename
func_name = <name>.__traceback__.tb_frame.f_code.co_name
line = linecache.getline(filename, <name>.__traceback__.tb_lineno)
trace_str = ''.join(traceback.format_tb(<name>.__traceback__))
error_msg = ''.join(traceback.format_exception(type(<name>), <name>, <name>.__traceback__))
```

Built-in Exceptions

BaseException	
├── SystemExit	# Raised by the sys.exit() function.
├── KeyboardInterrupt	# Raised when the user hits the interrupt key (ctrl-c).
├── Exception	# User-defined exceptions should be derived from this class.
│ ├── ArithmeticError	# Base class for arithmetic errors such as ZeroDivisionError
│ ├── AssertionError	# Raised by `assert <exp>` if expression returns false value
│ ├── AttributeError	# Raised when object doesn't have requested attribute/method
│ ├── EOFError	# Raised by input() when it hits an end-of-file condition.
│ ├── LookupError	# Base class for errors when a collection can't find an item
│ │ ├── IndexError	# Raised when a sequence index is out of range.
│ │ └── KeyError	# Raised when a dictionary key or set element is missing.
│ ├── MemoryError	# Out of memory. May be too late to start deleting variables
│ ├── NameError	# Raised when nonexistent name (variable/func/class) is used
│ │ └── UnboundLocalError	# Raised when local name is used before it's being defined.
│ ├── OSError	# Errors such as FileExistsError/TimeoutError (see #Open).
│ │ ├── ConnectionError	# Errors such as BrokenPipeError/ConnectionAbortedError.
│ ├── RuntimeError	# Raised by errors that don't fall into other categories.
│ │ ├── NotImplementedError	# Can be raised by abstract methods or by unfinished code.
│ │ └── RecursionError	# Raised when the maximum recursion depth is exceeded.
│ ├── StopIteration	# Raised when an empty iterator is passed to next().
│ ├── TypeError	# When an argument of the wrong type is passed to function.
│ └── ValueError	# When argument has the right type but inappropriate value.

Collections and their exceptions:

	List	Set	Dict
getitem()	IndexError		KeyError
pop()	IndexError	KeyError	KeyError
remove()	ValueError	KeyError	
index()	ValueError		

Useful built-in exceptions:

```
raise TypeError('Argument is of the wrong type!')
raise ValueError('Argument has the right type but an inappropriate value!')
raise RuntimeError('I am too lazy to define my own exception!')
```

User-defined Exceptions

```
class MyError(Exception): pass
class MyInputError(MyError): pass
```

Exit

Exits the interpreter by raising `SystemExit` exception.

```
import sys
sys.exit()                # Exits with exit code 0 (success).
sys.exit(<int>)           # Exits with the passed exit code.
sys.exit(<obj>)           # Prints to stderr and exits with 1.
```

Print

```
print(<el_1>, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Use **'file=sys.stderr'** for messages about errors.
Stdout and stderr streams hold output in a buffer until they receive a string containing `\n` or `\r`, buffer reaches 4096 characters, **'flush=True'** is used, or program exits.

Pretty Print

```
from pprint import pprint
pprint(<collection>, width=80, depth=None, compact=False, sort_dicts=True)
```

Each item is printed on its own line if collection exceeds 'width' characters.
Nested collections that are 'depth' levels deep get printed as '...'.

Input

```
<str> = input()
```

Reads a line from the user input or pipe if present (trailing newline gets stripped).
If argument is passed, it gets printed to the standard output before input is read.
`EOFError` is raised if user hits EOF (ctrl-d/ctrl-z↵) or input stream gets exhausted.

Command Line Arguments

```
import sys
scripts_path = sys.argv[0]
arguments    = sys.argv[1:]
```


Argument Parser

```
from argparse import ArgumentParser, FileType
p = ArgumentParser(description=<str>)
p.add_argument('-<short_name>', '--<name>', action='store_true') # Returns a parser.
p.add_argument('-<short_name>', '--<name>', type=<type>) # Flag (defaults to False).
p.add_argument('<name>', type=<type>, nargs=1) # Option (defaults to None).
p.add_argument('<name>', type=<type>, nargs='+') # Mandatory first argument.
p.add_argument('<name>', type=<type>, nargs='?/*') # Mandatory remaining args.
args = p.parse_args() # Optional argument/s.
obj = args.<name> # Exits on parsing error.
# Returns `<type>(<arg>)`.
```

Use **'help=<str>'** to set argument description that will be displayed in help message.

Use **'default=<obj>'** to set option's or optional argument's default value.

Use **'type=FileType(<mode>)'** for files. Accepts 'encoding', but 'newline' is None.

Open

Opens a file and returns the corresponding file object.

```
<file> = open(<path>, mode='r', encoding=None, newline=None)
```

'encoding=None' means that the default encoding is used, which is platform dependent. Best practice is to use **'encoding="utf-8"'** whenever possible.

'newline=None' means all different end of line combinations are converted to '\n' on read, while on write all '\n' characters are converted to system's default line separator.

'newline=""' means no conversions take place, but input is still broken into chunks by `readline()` and `readlines()` on every '\n', '\r' and '\r\n'.

Modes

- 'r'** - Read. Used by default.
- 'w'** - Write. Deletes existing contents.
- 'x'** - Write or fail if the file already exists.
- 'a'** - Append. Creates new file if it doesn't exist.
- 'w+'** - Read and write. Deletes existing contents.
- 'r+'** - Read and write from the start.
- 'a+'** - Read and write from the end.
- 'b'** - Binary mode (**'rb'**, **'wb'**, **'xb'**, ...).

Exceptions

'FileNotFoundError' can be raised when reading with **'r'** or **'r+'**.

'FileExistsError' can be raised when writing with **'x'**.

'IsADirectoryError' and **'PermissionError'** can be raised by any.

'OSError' is the parent class of all listed exceptions.

File Object

```
<file>.seek(0) # Moves to the start of the file.
<file>.seek(offset) # Moves 'offset' chars/bytes from the start.
<file>.seek(0, 2) # Moves to the end of the file.
<bin_file>.seek(±offset, origin) # Origin: 0 start, 1 current position, 2 end.
```

```
<str/bytes> = <file>.read(size=-1) # Reads 'size' chars/bytes or until EOF.
<str/bytes> = <file>.readline() # Returns a line or empty string/bytes on EOF.
<list> = <file>.readlines() # Returns a list of remaining lines.
<str/bytes> = next(<file>) # Returns a line using buffer. Do not mix.
```

```
<file>.write(<str/bytes>) # Writes a string or bytes object.
<file>.writelines(<collection>) # Writes a coll. of strings or bytes objects.
<file>.flush() # Flushes write buffer. Runs every 4096/8192 B.
<file>.close() # Closes the file after flushing write buffer.
```

Methods do not add or strip trailing newlines, not even writelines().

Read Text from File

```
def read_file(filename):  
    with open(filename, encoding='utf-8') as file:  
        return file.readlines()
```

Write Text to File

```
def write_to_file(filename, text):  
    with open(filename, 'w', encoding='utf-8') as file:  
        file.write(text)
```

Paths

```
import os, glob  
from pathlib import Path
```

```
<str> = os.getcwd()           # Returns working dir. Starts as shell's $PWD.  
<str> = os.path.join(<path>, ...) # Joins two or more pathname components.  
<str> = os.path.realpath(<path>) # Resolves symlinks and calls path.abspath().
```

```
<str> = os.path.basename(<path>) # Returns final component of the path.  
<str> = os.path.dirname(<path>)  # Returns path without the final component.  
<tuple> = os.path.splitext(<path>) # Splits on last period of the final component.
```

```
<list> = os.listdir(path='.')    # Returns filenames located at the path.  
<list> = glob.glob('<pattern>')  # Returns paths matching the wildcard pattern.
```

```
<bool> = os.path.exists(<path>)  # Or: <Path>.exists()  
<bool> = os.path.isfile(<path>)  # Or: <DirEntry/Path>.is_file()  
<bool> = os.path.isdir(<path>)   # Or: <DirEntry/Path>.is_dir()
```

```
<stat> = os.stat(<path>)          # Or: <DirEntry/Path>.stat()  
<num> = <stat>.st_mtime/st_size/... # Modification time, size in bytes, etc.
```

DirEntry

Unlike `listdir()`, `scandir()` returns `DirEntry` objects that cache `isfile`, `isdir`, and on Windows also `stat` information, thus significantly increasing the performance of code that requires it.

```
<iter> = os.scandir(path='.')    # Returns DirEntry objects located at the path.  
<str> = <DirEntry>.path          # Returns the whole path as a string.  
<str> = <DirEntry>.name          # Returns final component as a string.  
<file> = open(<DirEntry>)        # Opens the file and returns a file object.
```

Path Object

```
<Path> = Path(<path> [, ...])    # Accepts strings, Paths, and DirEntry objects.  
<Path> = <path> / <path> [/ ...] # First or second path must be a Path object.  
<Path> = <Path>.resolve()        # Returns absolute path with resolved symlinks.
```

```
<Path> = Path()                  # Returns relative CWD. Also Path('.').  
<Path> = Path.cwd()              # Returns absolute CWD. Also Path().resolve().  
<Path> = Path.home()             # Returns user's home directory (absolute).  
<Path> = Path(__file__).resolve() # Returns script's path if CWD wasn't changed.
```

<code><Path> = <Path>.parent</code>	<code># Returns Path without the final component.</code>
<code><str> = <Path>.name</code>	<code># Returns final component as a string.</code>
<code><str> = <Path>.stem</code>	<code># Returns final component without extension.</code>
<code><str> = <Path>.suffix</code>	<code># Returns final component's extension.</code>
<code><tup.> = <Path>.parts</code>	<code># Returns all components as strings.</code>
<code><iter> = <Path>.iterdir()</code>	<code># Returns directory contents as Path objects.</code>
<code><iter> = <Path>.glob('<pattern>')</code>	<code># Returns Paths matching the wildcard pattern.</code>
<code><str> = str(<Path>)</code>	<code># Returns path as a string.</code>
<code><file> = open(<Path>)</code>	<code># Also <Path>.read/write_text/bytes(<args>).</code>

OS Commands

```
import os, shutil, subprocess
```

<code>os.chdir(<path>)</code>	<code># Changes the current working directory.</code>
<code>os.mkdir(<path>, mode=0o777)</code>	<code># Creates a directory. Permissions are in octal.</code>
<code>os.makedirs(<path>, mode=0o777)</code>	<code># Creates all path's dirs. Also `exist_ok=False`.</code>

<code>shutil.copy(from, to)</code>	<code># Copies the file. 'to' can exist or be a dir.</code>
<code>shutil.copy2(from, to)</code>	<code># Also copies creation and modification time.</code>
<code>shutil.copytree(from, to)</code>	<code># Copies the directory. 'to' must not exist.</code>

<code>os.rename(from, to)</code>	<code># Renames/moves the file or directory.</code>
<code>os.replace(from, to)</code>	<code># Same, but overwrites file 'to' even on Windows.</code>
<code>shutil.move(from, to)</code>	<code># Rename() that moves into 'to' if it's a dir.</code>

<code>os.remove(<path>)</code>	<code># Deletes the file.</code>
<code>os.rmdir(<path>)</code>	<code># Deletes the empty directory.</code>
<code>shutil.rmtree(<path>)</code>	<code># Deletes the directory.</code>

Paths can be either strings, Paths, or DirEntry objects.

Functions report OS related errors by raising either `OSError` or one of its subclasses.

Shell Commands

<code><pipe> = os.popen('<commands>')</code>	<code># Executes commands in sh/cmd. Returns combined stdout.</code>
<code><str> = <pipe>.read(size=-1)</code>	<code># Reads 'size' chars or until EOF. Also readline/s().</code>
<code><int> = <pipe>.close()</code>	<code># Returns None if last command exited with returncode 0.</code>

Sends '1 + 1' to the basic calculator and captures its output:

```
>>> subprocess.run('bc', input='1 + 1\n', capture_output=True, text=True)
CompletedProcess(args='bc', returncode=0, stdout='2\n', stderr='')

```

Sends test.in to the basic calculator running in standard mode and saves its output to test.out:

```
>>> from shlex import split
>>> os.popen('echo 1 + 1 > test.in')
>>> subprocess.run(split('bc -s'), stdin=open('test.in'), stdout=open('test.out', 'w'))
CompletedProcess(args=['bc', '-s'], returncode=0)
>>> open('test.out').read()
'2\n'

```

JSON

Text file format for storing collections of strings and numbers.

<code>import json</code>	
<code><str> = json.dumps(<list/dict>)</code>	<code># Converts collection to JSON string.</code>
<code><coll> = json.loads(<str>)</code>	<code># Converts JSON string to collection.</code>

Read Collection from JSON File

```
def read_json_file(filename):  
    with open(filename, encoding='utf-8') as file:  
        return json.load(file)
```

Write Collection to JSON File

```
def write_to_json_file(filename, collection):  
    with open(filename, 'w', encoding='utf-8') as file:  
        json.dump(collection, file, ensure_ascii=False, indent=2)
```

Pickle

Binary file format for storing Python objects.

```
import pickle  
<bytes> = pickle.dumps(<object>)    # Converts object to bytes object.  
<object> = pickle.loads(<bytes>)    # Converts bytes object to object.
```

Read Object from Pickle File

```
def read_pickle_file(filename):  
    with open(filename, 'rb') as file:  
        return pickle.load(file)
```

Write Object to Pickle File

```
def write_to_pickle_file(filename, an_object):  
    with open(filename, 'wb') as file:  
        pickle.dump(an_object, file)
```

CSV

Text file format for storing spreadsheets.

```
import csv
```

Read

```
<reader> = csv.reader(<file>)        # Also: `dialect='excel', delimiter=','.  
<list>    = next(<reader>)            # Returns next row as a list of strings.  
<list>    = list(<reader>)            # Returns a list of remaining rows.
```

File must be opened with a **'newline=""'** argument, or newlines embedded inside quoted fields will not be interpreted correctly!

To print the spreadsheet to the console use **Tabulate** library.

For XML and binary Excel files (xlsx, xlsxm and xlsb) use **Pandas** library.

Reader accepts any iterator of strings, not just files.

Write

```
<writer> = csv.writer(<file>)          # Also: `dialect='excel', delimiter=','.  
<writer>.writerow(<collection>)        # Encodes objects using `str(<el>)`.  
<writer>.writerows(<coll_of_coll>)    # Appends multiple rows.
```

File must be opened with a **'newline=""'** argument, or **'\r'** will be added in front of every **'\n'** on platforms that use **'\r\n'** line endings!

Open existing file with **'mode="a"'** to append to it or **'mode="w"'** to overwrite it.

Parameters

'dialect' - Master parameter that sets the default values. String or a 'csv.Dialect' object.

'delimiter' - A one-character string used to separate fields.

'lineterminator' - How writer terminates rows. Reader looks for '\n', '\r' and '\r\n'.

'quotechar' - Character for quoting fields containing delimiters, quotechars, '\n' or '\r'.

'escapechar' - Character for escaping quotechars.

'doublequote' - Whether quotechars inside fields are/get doubled or escaped.

'quoting' - 0: As necessary, 1: All, 2: All but numbers which are read as floats, 3: None.

'skipinitialspace' - Is space character at the start of the field stripped by the reader.

Dialects

	excel	excel-tab	unix
delimiter	','	'\t'	','
lineterminator	'\r\n'	'\r\n'	'\n'
quotechar	'\"'	'\"'	'\"'
escapechar	None	None	None
doublequote	True	True	True
quoting	0	0	1
skipinitialspace	False	False	False

Read Rows from CSV File

```
def read_csv_file(filename, **csv_params):  
    with open(filename, encoding='utf-8', newline='') as file:  
        return list(csv.reader(file, **csv_params))
```

Write Rows to CSV File

```
def write_to_csv_file(filename, rows, mode='w', **csv_params):  
    with open(filename, mode, encoding='utf-8', newline='') as file:  
        writer = csv.writer(file, **csv_params)  
        writer.writerows(rows)
```

SQLite

A server-less database engine that stores each database into its own file.

```
import sqlite3  
<conn> = sqlite3.connect(<path>) # Opens existing or new file. Also ':memory:'.  
<conn>.close() # Closes connection. Discards uncommitted data.
```

Read

```
<cursor> = <conn>.execute('<query>') # Can raise a subclass of sqlite3.Error.  
<tuple> = <cursor>.fetchone() # Returns next row. Also next(<cursor>).  
<list> = <cursor>.fetchall() # Returns remaining rows. Also list(<cursor>).
```

Write

```
<conn>.execute('<query>') # Can raise a subclass of sqlite3.Error.  
<conn>.commit() # Saves all changes since the last commit.  
<conn>.rollback() # Discards all changes since the last commit.
```

Or:

```
with <conn>:                                     # Exits the block with commit() or rollback(),
    <conn>.execute('<query>')                     # depending on whether any exception occurred.
```

Placeholders

```
<conn>.execute('<query>', <list/tuple>)           # Replaces '?'s in query with values.
<conn>.execute('<query>', <dict/namedtuple>)       # Replaces ':<key>'s with values.
<conn>.executemany('<query>', <coll_of_coll>)      # Runs execute() multiple times.
```

Passed values can be of type str, int, float, bytes, None, or bool (stored as 1 or 0).

Example

Values are not actually saved in this example because `'conn.commit()'` is omitted!

```
>>> conn = sqlite3.connect('test.db')
>>> conn.execute('CREATE TABLE person (person_id INTEGER PRIMARY KEY, name, height)')
>>> conn.execute('INSERT INTO person VALUES (NULL, ?, ?)', ('Jean-Luc', 187)).lastrowid
1
>>> conn.execute('SELECT * FROM person').fetchall()
[(1, 'Jean-Luc', 187)]
```

SQLAlchemy

Library for interacting with various DB systems via SQL, method chaining, or ORM.

```
# $ pip3 install sqlalchemy
from sqlalchemy import create_engine, text
<engine> = create_engine('<url>')                # Url: 'dialect://user:password@host/dbname'.
<conn>    = <engine>.connect()                  # Creates a connection. Also <conn>.close().
<cursor>  = <conn>.execute(text('<query>'), ...) # '<dict>'. Replaces ':<key>'s with values.
with <conn>.begin(): ...                        # Exits the block with commit or rollback.
```

Dialect	pip3 install	Dependencies
mysql	mysqlclient	www.pypi.org/project/mysqlclient
postgresql	psycopg2	www.pypi.org/project/psycopg2
mssql	pyodbc	www.pypi.org/project/pyodbc
oracle+oracledb	oracledb	www.pypi.org/project/oracledb

Bytes

A bytes object is an immutable sequence of single bytes. Mutable version is called bytearray.

```
<bytes> = b'<str>'                             # Only accepts ASCII characters and \x00-\xff.
<int>    = <bytes>[index]                       # Returns an int in range from 0 to 255.
<bytes>  = <bytes>[<slice>]                     # Returns bytes even if it has only one element.
<bytes>  = <bytes>.join(<coll_of_bytes>)        # Joins elements using bytes as a separator.
```

Encode

```
<bytes> = bytes(<coll_of_ints>)                 # Ints must be in range from 0 to 255.
<bytes> = bytes(<str>, 'utf-8')                 # Encodes the string. Also <str>.encode().
<bytes> = bytes.fromhex('<hex>')                # Hex pairs can be separated by whitespaces.
<bytes> = <int>.to_bytes(n_bytes, ...)          # 'byteorder='big/little', signed=False'.
```

Decode

```
<list> = list(<bytes>)           # Returns ints in range from 0 to 255.
<str>  = str(<bytes>, 'utf-8')   # Returns a string. Also <bytes>.decode().
<str>  = <bytes>.hex()          # Returns hex pairs. Accepts `sep=<str>`.
<int>  = int.from_bytes(<bytes>, ...) # `byteorder='big/little', signed=False`.
```

Read Bytes from File

```
def read_bytes(filename):
    with open(filename, 'rb') as file:
        return file.read()
```

Write Bytes to File

```
def write_bytes(filename, bytes_obj):
    with open(filename, 'wb') as file:
        file.write(bytes_obj)
```

Struct

Module that performs conversions between a sequence of numbers and a bytes object.
System's type sizes, byte order, and alignment rules are used by default.

```
from struct import pack, unpack

<bytes> = pack('<format>', <el_1> [, ...]) # Packs objects according to format string.
<tuple> = unpack('<format>', <bytes>)      # Use iter_unpack() to get iterator of tuples.

>>> pack('>hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('>hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
```

Format

For standard type sizes and manual alignment (padding) start format string with:

- '=' - System's byte order (usually little-endian).
- '<' - Little-endian (i.e. least significant byte first).
- '>' - Big-endian (also '!').

Besides numbers, pack() and unpack() also support bytes objects as a part of the sequence:

- 'c' - A bytes object with a single element. For pad byte use 'x'.
- '<n>s' - A bytes object with n elements (not effected by byte order).

Integer types. Use a capital letter for unsigned type. Minimum and standard sizes are in brackets:

- 'b' - char (1/1)
- 'h' - short (2/2)
- 'i' - int (2/4)
- 'l' - long (4/4)
- 'q' - long long (8/8)

Floating point types (struct always uses standard sizes):

- 'f' - float (4/4)
- 'd' - double (8/8)

Array

List that can only hold numbers of a predefined type. Available types and their minimum sizes in bytes are listed above. Type sizes and byte order are always determined by the system, however bytes of each element can be reversed with `byteswap()` method.

```
from array import array
```

```
<array> = array('<typecode>', <coll_of_nums>) # Creates array from collection of numbers.
<array> = array('<typecode>', <bytes>)          # Writes passed bytes to array's memory.
<array> = array('<typecode>', <array>)          # Treats passed array as a sequence of numbers.
<array>.fromfile(<file>, n_items)             # Appends file's contents to array's memory.

<bytes> = bytes(<array>)                      # Returns a copy of array's memory.
<file>.write(<array>)                         # Writes array's memory to the binary file.
```

Memory View

A sequence object that points to the memory of another bytes-like object. Each element can reference a single or multiple consecutive bytes, depending on format. Order and number of elements can be changed with slicing.

```
<mview> = memoryview(<bytes/bytearray/array>) # Immutable if bytes is passed, else mutable.
<obj>    = <mview>[index]                     # Returns int or float. Bytes if format is 'c'.
<mview> = <mview>[<slice>]                     # Returns memoryview with rearranged elements.
<mview> = <mview>.cast('<typecode>')           # Only works between B/b/c and other types.
<mview>.release()                             # Releases memory buffer of the base object.

<bytes> = bytes(<mview>)                      # Returns a new bytes object. Also bytearray().
<bytes> = <bytes>.join(<coll_of_mviews>)       # Joins memoryviews using bytes as a separator.
<array> = array('<typecode>', <mview>)          # Treats memoryview as a sequence of numbers.
<file>.write(<mview>)                         # Writes `bytes(<mview>)` to the binary file.

<list>  = list(<mview>)                      # Returns a list of ints, floats, or bytes.
<str>   = str(<mview>, 'utf-8')               # Treats memoryview as a bytes object.
<str>   = <mview>.hex()                      # Returns hex pairs. Accepts `sep=<str>`.
```

Deque

List with efficient appends and pops from either side.

```
from collections import deque
```

```
<deque> = deque(<collection>)                 # Use `maxlen=<int>` to set size limit.
<deque>.appendleft(<el>)                     # Opposite element is dropped if full.
<deque>.extendleft(<collection>)              # Passed collection gets reversed.
<deque>.rotate(n=1)                          # Last element becomes first.
<el> = <deque>.popleft()                      # Raises IndexError if deque is empty.
```

Operator

Module of functions that provide the functionality of operators. Functions are ordered and grouped by operator precedence, from least to most binding. Logical and arithmetic operators in lines 1, 3 and 5 are also ordered by precedence within their own group.

```
import operator as op
```


<bool> = op.not_(<obj>)	# or, and, not (or/and missing)
<bool> = op.eq/ne/lt/ge/is_/is_not/contains(<obj>, <obj>)	# ==, !=, <, >=, is, is not, in
<obj> = op.or_/xor/and_(<int/set>, <int/set>)	# , ^, &
<int> = op.lshift/rshift(<int>, <int>)	# <<, >>
<obj> = op.add/sub/mul/truediv/floordiv/mod(<obj>, <obj>)	# +, -, *, /, //, %
<num> = op.neg/invert(<num>)	# -, ~
<num> = op.pow(<num>, <num>)	# **
<func> = op.itemgetter/attrgetter/methodcaller(<obj> [, ...])	# [index/key], .name, .name([...])

```

elementwise_sum = map(op.add, list_a, list_b)
sorted_by_second = sorted(<coll>, key=op.itemgetter(1))
sorted_by_both = sorted(<coll>, key=op.itemgetter(1, 0))
first_element = op.methodcaller('pop', 0)(<list>)

```

Most operators call the object's special method that is named after them (second object is passed as an argument), while logical operators call their own code that relies on bool().

Comparisons can be chained: 'x < y < z' gets converted to '(x < y) and (y < z)'.

Match Statement

Executes the first block with matching pattern. Added in Python 3.10.

```

match <object/expression>:
    case <pattern> [if <condition>]:
        <code>
    ...

```

Patterns

<value_pattern> = 1/'abc'/True/None/math.pi	# Matches the literal or a dotted name.
<class_pattern> = <type>()	# Matches any object of that type (or ABC).
<wildcard_patt> = _	# Matches any object. Useful in last case.
<capture_patt> = <name>	# Matches any object and binds it to name.
<as_pattern> = <pattern> as <name>	# Binds match to name. Also <type>(<name>).
<or_pattern> = <pattern> <pattern> [...]	# Matches any of the patterns.
<sequence_patt> = [<pattern>, ...]	# Matches sequence with matching items.
<mapping_patt> = {<value_pattern>: <patt>, ...}	# Matches dictionary with matching items.
<class_pattern> = <type>(<attr_name>=<patt>, ...)	# Matches object with matching attributes.

Sequence pattern can also be written as a tuple.

Use '*<name>' and '**<name>' in sequence/mapping patterns to bind remaining items.

Sequence pattern must match all items of the collection, while mapping pattern does not.

Patterns can be surrounded with brackets to override precedence ('|' > 'as' > ',').

Built-in types allow a single positional pattern that is matched against the entire object.

All names that are bound in the matching case, as well as variables initialized in its block, are visible after the match statement.

Example

```

>>> from pathlib import Path
>>> match Path('/home/gto/python-cheatsheet/README.md'):
...     case Path(
...         parts=['/', 'home', user, *_]
...     ) as p if p.name.lower().startswith('readme') and p.is_file():
...         print(f'{p.name} is a readme file that belongs to user {user}.')
README.md is a readme file that belongs to user gto.

```

Logging

```
import logging as log
```

```
log.basicConfig(filename=<path>, level='DEBUG') # Configures the root logger (see Setup).
log.debug/info/warning/error/critical(<str>)   # Sends message to the root logger.
<Logger> = log.getLogger(__name__)             # Returns logger named after the module.
<Logger>.<level>(<str>)                         # Sends message to the logger.
<Logger>.exception(<str>)                     # Error() that appends caught exception.
```

Setup

```
log.basicConfig(
    filename=None, # Logs to stderr or appends to file.
    format='%(levelname)s: %(name)s: %(message)s', # Add '%(asctime)s' for local datetime.
    level=log.WARNING, # Drops messages with lower priority.
    handlers=[log.StreamHandler(sys.stderr)] # Uses FileHandler if filename is set.
)

<Formatter> = log.Formatter('<format>') # Creates a Formatter.
<Handler> = log.FileHandler(<path>, mode='a') # Creates a Handler. Also `encoding=None`.
<Handler>.setFormatter(<Formatter>) # Adds Formatter to the Handler.
<Handler>.setLevel(<int/str>) # Processes all messages by default.
<Logger>.addHandler(<Handler>) # Adds Handler to the Logger.
<Logger>.setLevel(<int/str>) # What is sent to its/ancestors' handlers.
<Logger>.propagate = <bool> # Cuts off ancestors' handlers if False.
```

Parent logger can be specified by naming the child logger '**<parent>.<name>**'.

If logger doesn't have a set level, it inherits it from the first ancestor that does.

Formatter also accepts: pathname, filename, funcName, lineno, thread and process.

RotatingFileHandler creates and deletes files based on 'maxBytes', 'backupCount' args.

An object with '**filter(<LogRecord>)**' method (or the method itself) can be added to loggers and handlers via addFilter(). Message is dropped if filter() returns a false value.

Creates a logger that writes all messages to a file and sends them to the root's handler that prints warnings or higher:

```
>>> logger = log.getLogger('my_module')
>>> handler = log.FileHandler('test.log', encoding='utf-8')
>>> handler.setFormatter(log.Formatter('%(asctime)s %(levelname)s: %(name)s: %(message)s'))
>>> logger.addHandler(handler)
>>> logger.setLevel('DEBUG')
>>> log.basicConfig()
>>> log.root.handlers[0].setLevel('WARNING')
>>> logger.critical('Running out of disk space.')
CRITICAL:my_module:Running out of disk space.
>>> print(open('test.log').read())
2023-02-07 23:21:01,430 CRITICAL:my_module:Running out of disk space.
```

Introspection

```
<list> = dir() # Local names of variables, functions, classes and modules
<dict> = vars() # Dict of local names and their objects. Also locals().
<dict> = globals() # Dict of global names and their objects, e.g. __builtin__
```

```
<list> = dir(<obj>) # Returns names of object's attributes (including methods)
<dict> = vars(<obj>) # Returns dict of writable attributes. Also <obj>.__dict__
<bool> = hasattr(<obj>, '<name>') # Checks if object possesses attribute with passed name.
value = getattr(<obj>, '<name>') # Returns object's attribute or raises AttributeError.
setattr(<obj>, '<name>', value) # Sets attribute. Only works on objects with __dict__ attr
delattr(<obj>, '<name>') # Deletes attribute from __dict__. Also `del <obj>.<name>`
```

```

<Sig> = inspect.signature(<func>) # Returns a Signature object of the passed function.
<dict> = <Sig>.parameters         # Returns dict of Parameters. Also <Sig>.return_annotation
<memb> = <Param>.kind              # Returns ParameterKind member (Parameter.KEYWORD_ONLY, ...)
<type> = <Param>.annotation        # Returns Parameter.empty if missing. Also <Param>.default

```

Threading

CPython interpreter can only run a single thread at a time. Using multiple threads won't result in a faster execution, unless at least one of the threads contains an I/O operation.

```

from threading import Thread, Lock, RLock, Semaphore, Event, Barrier
from concurrent.futures import ThreadPoolExecutor, as_completed

```

Thread

```

<Thread> = Thread(target=<function>) # Use `args=<collection>` to set the arguments.
<Thread>.start()                     # Starts the thread. Also <Thread>.is_alive().
<Thread>.join()                      # Waits for the thread to finish executing.

```

Use **'kwargs=<dict>'** to pass keyword arguments to the function.
 Use **'daemon=True'**, or the program will not be able to exit while the thread is alive.

Lock

```

<lock> = Lock/RLock()               # RLock can only be released by acquirer.
<lock>.acquire()                     # Waits for the lock to be available.
<lock>.release()                     # Makes the lock available again.

```

Or:

```

with <lock>:                         # Enters the block by calling acquire() and
    ...                             # exits it with release(), even on error.

```

Semaphore, Event, Barrier

```

<Semaphore> = Semaphore(value=1)    # Lock that can be acquired by 'value' threads.
<Event>      = Event()               # Method wait() blocks until set() is called.
<Barrier>    = Barrier(n_times)      # Wait() blocks until it's called n times.

```

Queue

```

<Queue> = queue.Queue(maxsize=0)    # A thread-safe first-in-first-out queue.
<Queue>.put(<el>)                    # Blocks until queue stops being full.
<Queue>.put_nowait(<el>)             # Raises queue.Full exception if full.
<el> = <Queue>.get()                 # Blocks until queue stops being empty.
<el> = <Queue>.get_nowait()           # Raises queue.Empty exception if empty.

```

Thread Pool Executor

```

<Exec> = ThreadPoolExecutor(max_workers=None) # Or: `with ThreadPoolExecutor() as <name>: ...
<iter> = <Exec>.map(<func>, <args_1>, ...)    # Multithreaded and non-lazy map(). Keeps order
<Futr> = <Exec>.submit(<func>, <arg_1>, ...) # Creates a thread and returns its Future obj.
<Exec>.shutdown()                           # Blocks until all threads finish executing.

```

```

<bool> = <Future>.done()                # Checks if the thread has finished executing.
<obj> = <Future>.result(timeout=None)     # Waits for thread to finish and returns result
<bool> = <Future>.cancel()               # Cancels or returns False if running/finished.
<iter> = as_completed(<coll_of_Futures>) # `next(<iter>)` returns next completed Future.

```

Map() and as_completed() also accept 'timeout'. It causes futures.TimeoutError when next() is called/blocking. Map() times from original call and as_completed() from

first call to next(). As_completed() fails if next() is called too late, even if all threads are done.

Exceptions that happen inside threads are raised when map iterator's next() or Future's result() are called. Future's exception() method returns exception object or None.

ProcessPoolExecutor provides true parallelism but: everything sent to/from workers must be **pickable**, queues must be sent using executor's 'initargs' and 'initializer' parameters, and executor should only be reachable via **'if __name__ == "__main__": ...'**.

Coroutines

Coroutines have a lot in common with threads, but unlike threads, they only give up control when they call another coroutine and they don't use as much memory.

Coroutine definition starts with **'async'** and its call with **'await'**.

Use **'asyncio.run(<coroutine>')** to start the first/main coroutine.

```
import asyncio as aio
```

<coro> = <async_function>(<args>)	# Creates a coroutine by calling async def function
<obj> = await <coroutine>	# Starts the coroutine and returns its result.
<task> = aio.create_task(<coroutine>)	# Schedules the coroutine for execution.
<obj> = await <task>	# Returns coroutine's result. Also <task>.cancel().
<coro> = aio.gather(<coro/task>, ...)	# Schedules coros. Returns list of results on await
<coro> = aio.wait(<tasks>, return_when=...)	# 'ALL/FIRST_COMPLETED' . Returns (done, pending).
<iter> = aio.as_completed(<coros/tasks>)	# Iter of coros that return next result on await.

Runs a terminal game where you control an asterisk that must avoid numbers:

```
import asyncio, collections, curses, curses.textpad, enum, random

P = collections.namedtuple('P', 'x y')      # Position
D = enum.Enum('D', 'n e s w')              # Direction
W, H = 15, 7                               # Width, Height

def main(screen):
    curses.curs_set(0)                      # Makes cursor invisible.
    screen.nodelay(True)                   # Makes getch() non-blocking.
    asyncio.run(main_coroutine(screen))    # Starts running asyncio code.

async def main_coroutine(screen):
    moves = asyncio.Queue()
    state = {'*': P(0, 0)} | {id_: P(W//2, H//2) for id_ in range(10)}
    ai    = [random_controller(id_, moves) for id_ in range(10)]
    mvc   = [human_controller(screen, moves), model(moves, state), view(state, screen)]
    tasks = [asyncio.create_task(coro) for coro in ai + mvc]
    await asyncio.wait(tasks, return_when=asyncio.FIRST_COMPLETED)

async def random_controller(id_, moves):
    while True:
        d = random.choice(list(D))
        moves.put_nowait((id_, d))
        await asyncio.sleep(random.triangular(0.01, 0.65))

async def human_controller(screen, moves):
    while True:
        key_mappings = {258: D.s, 259: D.n, 260: D.w, 261: D.e}
        if d := key_mappings.get(screen.getch()):
            moves.put_nowait(( '*', d))
            await asyncio.sleep(0.005)

async def model(moves, state):
    while state['*'] not in (state[id_] for id_ in range(10)):
        id_, d = await moves.get()
        deltas = {D.n: P(0, -1), D.e: P(1, 0), D.s: P(0, 1), D.w: P(-1, 0)}
        state[id_] = P((state[id_].x + deltas[d].x) % W, (state[id_].y + deltas[d].y) % H)

async def view(state, screen):
    offset = P(curses.COLS//2 - W//2, curses.LINES//2 - H//2)
    while True:
        screen.erase()
        curses.textpad.rectangle(screen, offset.y-1, offset.x-1, offset.y+H, offset.x+W)
        for id_, p in state.items():
            screen.addstr(offset.y + (p.y - state['*'].y + H//2) % H,
                          offset.x + (p.x - state['*'].x + W//2) % W, str(id_))
        screen.refresh()
        await asyncio.sleep(0.005)

if __name__ == '__main__':
    curses.wrapper(main)
```

Libraries

Progress Bar

```
# $ pip3 install tqdm
>>> import tqdm, time
>>> for el in tqdm.tqdm([1, 2, 3], desc='Processing'):
...     time.sleep(1)
Processing: 100%|██████████| 3/3 [00:03<00:00, 1.00s/it]
```

Plot

```
# $ pip3 install matplotlib
import matplotlib.pyplot as plt

plt.plot/bar/scatter(x_data, y_data [, label=<str>]) # Also plt.plot(y_data).
plt.legend() # Adds a legend.
plt.title/xlabel/ylabel(<str>) # Adds a title or label.
plt.savefig(<path>) # Saves the plot.
plt.show() # Displays the plot.
plt.clf() # Clears the plot.
```

Table

Prints a CSV spreadsheet to the console:

```
# $ pip3 install tabulate
import csv, tabulate
with open('test.csv', encoding='utf-8', newline='') as file:
    rows = list(csv.reader(file))
print(tabulate.tabulate(rows, headers='firstrow'))
```

Console App

Runs a basic file explorer in the console:

```
# $ pip3 install windows-curses
import curses, os
from curses import A_REVERSE, KEY_DOWN, KEY_UP, KEY_LEFT, KEY_RIGHT, KEY_ENTER

def main(screen):
    ch, first, selected, paths = 0, 0, 0, os.listdir()
    while ch != ord('q'):
        height, width = screen.getmaxyx()
        screen.erase()
        for y, filename in enumerate(paths[first : first+height]):
            color = A_REVERSE if filename == paths[selected] else 0
            screen.addnstr(y, 0, filename, width-1, color)
        ch = screen.getch()
        selected += (ch == KEY_DOWN) - (ch == KEY_UP)
        selected = max(0, min(len(paths)-1, selected))
        first += (selected >= first + height) - (selected < first)
        if ch in [KEY_LEFT, KEY_RIGHT, KEY_ENTER, ord('\n'), ord('\r')]:
            new_dir = '..' if ch == KEY_LEFT else paths[selected]
            if os.path.isdir(new_dir):
                os.chdir(new_dir)
                first, selected, paths = 0, 0, os.listdir()

if __name__ == '__main__':
    curses.wrapper(main)
```

GUI App

A weight converter GUI application:

```
# $ pip3 install PySimpleGUI
import PySimpleGUI as sg

text_box = sg.Input(default_text='100', enable_events=True, key='-QUANTITY-')
dropdown = sg.InputCombo(['g', 'kg', 't'], 'kg', readonly=True, enable_events=True, k='-UNIT-')
label = sg.Text('100 kg is 220.462 lbs.', key='-OUTPUT-')
button = sg.Button('Close')
window = sg.Window('Weight Converter', [[text_box, dropdown], [label], [button]])

while True:
    event, values = window.read()
    if event in [sg.WIN_CLOSED, 'Close']:
        break
    try:
        quantity = float(values['-QUANTITY-'])
    except ValueError:
        continue
    unit = values['-UNIT-']
    factors = {'g': 0.001, 'kg': 1, 't': 1000}
    lbs = quantity * factors[unit] / 0.45359237
    window['-OUTPUT-'].update(value=f'{quantity} {unit} is {lbs:g} lbs.')
window.close()
```

Scraping

Scrapes Python's URL and logo from its Wikipedia page:

```
# $ pip3 install requests beautifulsoup4
import requests, bs4, os

response = requests.get('https://en.wikipedia.org/wiki/Python_(programming_language)')
document = bs4.BeautifulSoup(response.text, 'html.parser')
table = document.find('table', class_='infobox vevent')
python_url = table.find('th', text='Website').next_sibling.a['href']
logo_url = table.find('img')['src']
filename = os.path.basename(logo_url)
with open(filename, 'wb') as file:
    file.write(requests.get(f'https://{logo_url}').content)
print(f'{python_url}, file://{os.path.abspath(filename)}')
```

Selenium

Library for scraping websites with dynamic content.

```
# $ pip3 install selenium
from selenium import webdriver

<WebDrv> = webdriver.Chrome/Firefox/Safari/Edge() # Opens a browser. Also <WebDrv>.quit().
<WebDrv>.get('<url>') # Also <WebDrv>.implicitly_wait(seconds)
<str> = <WebDrv>.page_source # Returns HTML of fully rendered page.
<El> = <WebDrv/El>.find_element('css selector', ...) # '<tag>#<id>.<class>[<attr>="<val>"]...'.
<list> = <WebDrv/El>.find_elements('xpath', ...) # '//<tag>[<attr>="<val>"]...'. See XPath
<str> = <El>.get_attribute(<str>) # Property if exists. Also <El>.text.
<El>.click/clear() # Also <El>.send_keys(<str>).
```

XPath – also available in lxml, Scrapy, and browser's console via '\$x("<xpath>")':

```
<xpath> = //<element>[ or // <element>] # /<child>, //<descendant>, ../<sibling>
<xpath> = //<element>/following::<element> # Next element. Also preceding/parent/...
<element> = <tag><conditions><index> # '<tag> = */a/...', '<index> = [1/2/...]'
<condition> = [<sub_cond> [and/or <sub_cond>]] # For negation use 'not(<sub_cond>)'
<sub_cond> = @<attr>["<val>"] # '<text>()=', '.=' match (complete) text.
<sub_cond> = contains(@<attr>, "<val>") # Is <val> a substring of attr's value?
<sub_cond> = [//]<element> # Has matching child? Descendant if //.
```

Web App

Flask is a micro web framework/server. If you just want to open a html file in a web browser use `'webbrowser.open(<path>')` instead.

```
# $ pip3 install flask
import flask as fl
```

```
app = fl.Flask(__name__) # Returns the app object. Put at the top.
app.run(host=None, port=None, debug=None) # Or: $ flask --app FILE run [--ARG[=VAL]]...
```

Starts the app at `'http://localhost:5000'`. Use `'host="0.0.0.0"'` to run externally.

Install a WSGI server like [Waitress](#) and a HTTP server such as [Nginx](#) for better security.

Debug mode restarts the app whenever script changes and displays errors in the browser.

Static Request

```
@app.route('/img/<path:filename>')
def serve_file(filename):
    return fl.send_from_directory('DIRNAME/', filename)
```

Dynamic Request

```
@app.route('/<sport>')
def serve_html(sport):
    return fl.render_template_string('<h1>{{title}}</h1>', title=sport)
```

`'fl.render_template(filename, <kwargs>')` renders a file located in `'templates'` dir.

`'fl.abort(<int>')` returns error code and `'return fl.redirect(<url>')` redirects.

`'fl.request.args[<str>']` returns parameter from query string (URL part right of '?').

`'fl.session[<str>] = <obj>'` stores session data. It requires secret key to be set at the startup with `'app.secret_key = <str>'`.

REST Request

```
@app.post('/<sport>/odds')
def serve_json(sport):
    team = fl.request.form['team']
    return {'team': team, 'odds': [2.09, 3.74, 3.68]}
```

Starts the app in its own thread and queries its REST API:

```
# $ pip3 install requests
>>> import threading, requests
>>> threading.Thread(target=app.run, daemon=True).start()
>>> url = 'http://localhost:5000/football/odds'
>>> response = requests.post(url, data={'team': 'arsenal f.c.'})
>>> response.json()
{'team': 'arsenal f.c.', 'odds': [2.09, 3.74, 3.68]}
```

Profiling

```
from time import perf_counter
start_time = perf_counter()
...
duration_in_seconds = perf_counter() - start_time
```


Timing a Snippet

```
>>> from timeit import timeit
>>> timeit('list(range(10000))', number=1000, globals=globals(), setup='pass')
0.19373
```

Profiling by Line

```
$ pip3 install line_profiler
$ echo '@profile
def main():
    a = list(range(10000))
    b = set(range(10000))
main()' > test.py
$ kernprof -lv test.py
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
1					@profile
2					def main():
3	1	253.4	253.4	32.2	a = list(range(10000))
4	1	534.1	534.1	67.8	b = set(range(10000))

Call and Flame Graphs

```
$ apt/brew install graphviz && pip3 install gprof2dot snakeviz # Or download installer.
$ tail --lines=+2 test.py > test.py # Removes first line.
$ python3 -m cProfile -o test.prof test.py # Runs built-in profiler.
$ gprof2dot --format=pstats test.prof | dot -T png -o test.png # Generates call graph.
$ xdg-open/open test.png # Displays call graph.
$ snakeviz test.prof # Displays flame graph.
```

Sampling and Memory Profilers

pip3 install	Target	How to run	Lines	Live
pyinstrument	CPU	pyinstrument test.py	×	×
py-spy	CPU	py-spy top -- python3 test.py	×	✓
scalene	CPU+Memory	scalene test.py	✓	×
memray	Memory	memray run --live test.py	✓	✓

NumPy

Array manipulation mini-language. It can run up to one hundred times faster than the equivalent Python code. An even faster alternative that runs on a GPU is called CuPy.

```
# $ pip3 install numpy
import numpy as np
```

```
<array> = np.array(<list/list_of_lists/...>) # Returns a 1d/2d/... NumPy array.
<array> = np.zeros/ones/empty(<shape>) # Also np.full(<shape>, <el>).
<array> = np.arange(from_inc, to_exc, ±step) # Also np.linspace(start, stop, len).
<array> = np.random.randint(from_inc, to_exc, <shape>) # Also np.random.random(<shape>).
```



```
<view> = <array>.reshape(<shape>) # Also `<array>.shape = <shape>`.
<array> = <array>.flatten() # Also `<view> = <array>.ravel()`.
<view> = <array>.transpose() # Or: <array>.T
```



```
<array> = np.copy/abs/sqrt/log/int64(<array>) # Returns new array of the same shape.
<array> = <array>.sum/max/mean/argmax/all(axis) # Aggregates specified dimension.
<array> = np.apply_along_axis(<func>, axis, <array>) # Func can return a scalar or array.
```

```

<array> = np.concatenate(<list_of_arrays>, axis=0)      # Links arrays along first axis (rows)
<array> = np.vstack/column_stack(<list_of_arrays>)      # Treats 1d arrays as rows or columns.
<array> = np.tile/repeat(<array>, <int/list> [, axis])  # Tiles array or repeats its elements.

```

Shape is a tuple of dimension sizes. A 100x50 RGB image has shape (50, 100, 3).

Axis is an index of a dimension. Leftmost dimension has index 0. Summing the RGB image along axis 2 will return a greyscale image with shape (50, 100).

Indexing

```

<el>      = <2d>[row_index, col_index]                  # 0r: <3d>[<int>, <int>, <int>]
<1d_view> = <2d>[row_index]                            # 0r: <3d>[<int>, <int>, <slice>]
<1d_view> = <2d>[:, col_index]                         # 0r: <3d>[<int>, <slice>, <int>]
<2d_view> = <2d>[from:to_row_i, from:to_col_i]         # 0r: <3d>[<int>, <slice>, <slice>]

<1d_array> = <2d>[row_indices, col_indices]            # 0r: <3d>[<int/1d>, <1d>, <1d>]
<2d_array> = <2d>[row_indices]                        # 0r: <3d>[<int/1d>, <1d>, <slice>]
<2d_array> = <2d>[:, col_indices]                    # 0r: <3d>[<int/1d>, <slice>, <1d>]
<2d_array> = <2d>[np.ix_(row_indices, col_indices)]   # 0r: <3d>[<int/1d/2d>, <2d>, <2d>]

<2d_bools> = <2d> > <el/1d/2d>                      # 1d object must have size of a row.
<1/2d_arr> = <2d>[<2d/1d_bools>]                    # 1d_bools must have size of a column.

```

'**:**' returns a slice of all dimension's indices. Omitted dimensions default to '**:**'.

Python converts '**obj[i, j]**' to '**obj[(i, j)]**'. This makes '**<2d>[row_i, col_i]**' and '**<2d>[row_indices]**' indistinguishable to NumPy if tuple of indices is passed!

Indexing with a slice and 1d object works the same as when using two slices (lines 4, 6, 7).

'**ix_([1, 2], [3, 4])**' returns '**[[1], [2]]**' and '**[[3, 4]]**'. Due to broadcasting rules, this is the same as using '**[[1, 1], [2, 2]]**' and '**[[3, 4], [3, 4]]**'.

Any value that is broadcastable to the indexed shape can be assigned to the selection.

Broadcasting

A set of rules by which NumPy functions operate on arrays of different shapes.

```

left = np.array([0.1, 0.6, 0.8])                      # `left.shape == (3,)`
right = np.array([[0.1], [0.6], [0.8]])               # `right.shape == (3, 1)`

```

1. If array shapes differ in length, left-pad the shorter shape with ones:

```

left = np.array([[0.1, 0.6, 0.8]])                    # `left.shape == (1, 3)`
right = np.array([[0.1], [0.6], [0.8]])               # `right.shape == (3, 1)`

```

2. If any dimensions differ in size, expand the ones that have size 1 by duplicating their elements:

```

left = np.array([[0.1, 0.6, 0.8],                    # `left.shape == (3, 3)`
                 [0.1, 0.6, 0.8],
                 [0.1, 0.6, 0.8]])

right = np.array([[0.1, 0.1, 0.1],                    # `right.shape == (3, 3)`
                  [0.6, 0.6, 0.6],
                  [0.8, 0.8, 0.8]])

```

Example

For each point returns index of its nearest point ([0.1, 0.6, 0.8] => [1, 2, 1]):

```
>>> print(points := np.array([0.1, 0.6, 0.8]))
[0.1 0.6 0.8]
>>> print(wrapped_points := points.reshape(3, 1))
[[0.1]
 [0.6]
 [0.8]]
>>> print(deltas := points - wrapped_points)
[[ 0.   0.5  0.7]
 [-0.5  0.   0.2]
 [-0.7 -0.2  0. ]]
>>> deltas[range(3), range(3)] = np.inf
>>> print(distances := np.abs(deltas))
[[inf 0.5 0.7]
 [0.5 inf 0.2]
 [0.7 0.2 inf]]
>>> print(distances.argmin(axis=1))
[1 2 1]
```

Image

```
# $ pip3 install pillow
from PIL import Image
```

```
<Image> = Image.new('<mode>', (width, height)) # Creates new image. Also `color=<int/tuple>`.
<Image> = Image.open(<path>) # Identifies format based on file's contents.
<Image> = <Image>.convert('<mode>') # Converts image to the new mode (see Modes).
<Image>.save(<path>) # Selects format based on extension (PNG/JPG...)
<Image>.show() # Displays image in default preview app.
```

```
<int/tup> = <Image>.getpixel((x, y)) # Returns pixel's value (its color).
<ImgCore> = <Image>.getdata() # Returns a flattened view of pixel values.
<Image>.putpixel((x, y), <int/tuple>) # Updates pixel's value. Clips passed int/s.
<Image>.putdata(<list/ImgCore>) # Updates pixels with a copy of the sequence.
<Image>.paste(<Image>, (x, y)) # Draws passed image at the specified location
```

```
<Image> = <Image>.filter(<Filter>) # Use ImageFilter.<name>(<args>) for Filter.
<Image> = <Enhance>.enhance(<float>) # Use ImageEnhance.<name>(<Image>) for Enhance
```

```
<array> = np.array(<Image>) # Creates a 2d/3d NumPy array from the image.
<Image> = Image.fromarray(np.uint8(<array>)) # Use <array>.clip(0, 255) to clip the values.
```

Modes

- 'L' - Lightness (greyscale image). Each pixel is an int between 0 and 255.
- 'RGB' - Red, green, blue (true color image). Each pixel is a tuple of three ints.
- 'RGBA' - RGB with alpha. Low alpha (i.e. forth int) makes pixel more transparent.
- 'HSV' - Hue, saturation, value. Three ints representing color in HSV color space.

Examples

Creates a PNG image of a rainbow gradient:

```
WIDTH, HEIGHT = 100, 100
n_pixels = WIDTH * HEIGHT
hues = (255 * i/n_pixels for i in range(n_pixels))
img = Image.new('HSV', (WIDTH, HEIGHT))
img.putdata([(int(h), 255, 255) for h in hues])
img.convert('RGB').save('test.png')
```

Adds noise to the PNG image and displays it:

```
from random import randint
add_noise = lambda value: max(0, min(255, value + randint(-20, 20)))
img = Image.open('test.png').convert('HSV')
img.putdata([(add_noise(h), s, v) for h, s, v in img.getdata()])
img.show()
```

Image Draw

```
from PIL import ImageDraw
<Draw> = ImageDraw.Draw(<Image>) # Object for adding 2D graphics to the image.
<Draw>.point((x, y)) # Draws a point. Truncates floats into ints.
<Draw>.line((x1, y1, x2, y2 [, ...])) # To get anti-aliasing use Image's resize().
<Draw>.arc((x1, y1, x2, y2), deg1, deg2) # Draws in clockwise dir. Also pieslice().
<Draw>.rectangle((x1, y1, x2, y2)) # Also rounded_rectangle(), regular_polygon().
<Draw>.polygon((x1, y1, x2, y2, ...)) # Last point gets connected to the first.
<Draw>.ellipse((x1, y1, x2, y2)) # To rotate use Image's rotate() and paste().
<Draw>.text((x, y), <str>, font=<Font>) # `<Font> = ImageFont.truetype(<path>, size)`.
```

Use '**fill=<color>**' to set the primary color.

Use '**width=<int>**' to set the width of lines or contours.

Use '**outline=<color>**' to set the color of the contours.

Color can be an int, tuple, '**#rrggbb[aa]**' string or a color name.

Animation

Creates a GIF of a bouncing ball:

```
# $ pip3 install imageio
from PIL import Image, ImageDraw
import imageio

WIDTH, HEIGHT, R = 126, 126, 10
frames = []
for velocity in range(1, 16):
    y = sum(range(velocity))
    frame = Image.new('L', (WIDTH, HEIGHT))
    draw = ImageDraw.Draw(frame)
    draw.ellipse((WIDTH/2-R, y, WIDTH/2+R, y+R*2), fill='white')
    frames.append(frame)
frames += reversed(frames[1:-1])
imageio.mimsave('test.gif', frames, duration=0.03)
```

Audio

```
import wave
```

```
<Wave> = wave.open('<path>') # Opens the WAV file for reading.
<int> = <Wave>.getframerate() # Returns number of frames per second.
<int> = <Wave>.getnchannels() # Returns number of samples per frame.
<int> = <Wave>.getsampwidth() # Returns number of bytes per sample.
<tuple> = <Wave>.getparams() # Returns namedtuple of all parameters.
<bytes> = <Wave>.readframes(nframes) # Returns all frames if -1 is passed.
```

```
<Wave> = wave.open('<path>', 'wb') # Creates/truncates a file for writing.
<Wave>.setframerate(<int>) # Pass 44100 for CD, 48000 for video.
<Wave>.setnchannels(<int>) # Pass 1 for mono, 2 for stereo.
<Wave>.setsampwidth(<int>) # Pass 2 for CD, 3 for hi-res sound.
<Wave>.setparams(<tuple>) # Tuple must contain all parameters.
<Wave>.writeframes(<bytes>) # Appends frames to the file.
```

Bytes object contains a sequence of frames, each consisting of one or more samples.

In a stereo signal, the first sample of a frame belongs to the left channel.

Each sample consists of one or more bytes that, when converted to an integer, indicate the displacement of a speaker membrane at a given moment. If sample width is one byte, then the integer should be encoded unsigned. For all other sizes, the integer should be encoded signed with little-endian byte order.

Sample Values

sampwidth	min	zero	max
1	0	128	255
2	-32768	0	32767
3	-8388608	0	8388607

Read Float Samples from WAV File

```
def read_wav_file(filename):
    def get_int(bytes_obj):
        an_int = int.from_bytes(bytes_obj, 'little', signed=(p.sampwidth != 1))
        return an_int - 128 * (p.sampwidth == 1)
    with wave.open(filename) as file:
        p = file.getparams()
        frames = file.readframes(-1)
        bytes_samples = (frames[i : i + p.sampwidth] for i in range(0, len(frames), p.sampwidth))
        return [get_int(b) / pow(2, (p.sampwidth * 8) - 1) for b in bytes_samples], p
```

Write Float Samples to WAV File

```
def write_to_wav_file(filename, samples_f, p=None, nchannels=1, sampwidth=2, framerate=44100):
    def get_bytes(a_float):
        a_float = max(-1, min(1 - 2e-16, a_float))
        a_float += (p.sampwidth == 1)
        a_float *= pow(2, (p.sampwidth * 8) - 1)
        return int(a_float).to_bytes(p.sampwidth, 'little', signed=(p.sampwidth != 1))
    if p is None:
        p = wave._wave_params(nchannels, sampwidth, framerate, 0, 'NONE', 'not compressed')
    with wave.open(filename, 'wb') as file:
        file.setparams(p)
        file.writeframes(b''.join(get_bytes(f) for f in samples_f))
```

Examples

Saves a 440 Hz sine wave to a mono WAV file:

```
from math import pi, sin
samples_f = (sin(i * 2 * pi * 440 / 44100) for i in range(100_000))
write_to_wav_file('test.wav', samples_f)
```

Adds noise to the WAV file:

```
from random import uniform
samples_f, params = read_wav_file('test.wav')
samples_f = (f + uniform(-0.05, 0.05) for f in samples_f)
write_to_wav_file('test.wav', samples_f, params)
```

Plays the WAV file:

```
# $ pip3 install simpleaudio
from simpleaudio import play_buffer
with wave.open('test.wav') as file:
    p = file.getparams()
    frames = file.readframes(-1)
    play_buffer(frames, p.nchannels, p.sampwidth, p.framerate).wait_done()
```

Text to Speech

```
# $ pip3 install pyttsx3
import pyttsx3
engine = pyttsx3.init()
engine.say('Sally sells seashells by the seashore.')
engine.runAndWait()
```

Synthesizer

Plays Popcorn by Gershon Kingsley:

```
# $ pip3 install simpleaudio
import array, itertools as it, math, simpleaudio

F = 44100
P1 = '71J,69J,,71J,66J,,62J,66J,,59J,,,71J,69J,,71J,66J,,62J,66J,,59J,,, '
P2 = '71J,73J,,74J,73J,,74J,,71J,,73J,71J,,73J,,69J,,71J,69J,,71J,,67J,,71J,,, '
get_pause = lambda seconds: it.repeat(0, int(seconds * F))
sin_f = lambda i, hz: math.sin(i * 2 * math.pi * hz / F)
get_wave = lambda hz, seconds: (sin_f(i, hz) for i in range(int(seconds * F)))
get_hz = lambda note: 440 * 2 ** ((int(note[:2]) - 69) / 12)
get_sec = lambda note: 1/4 if 'J' in note else 1/8
get_samples = lambda note: get_wave(get_hz(note), get_sec(note)) if note else get_pause(1/8)
samples_f = it.chain.from_iterable(get_samples(n) for n in (P1+P2).split(','))
samples_i = array.array('h', (int(f * 30000) for f in samples_f))
simpleaudio.play_buffer(samples_i, 1, 2, F).wait_done()
```

Pygame

Opes a window and draws a square that can be moved with arrow keys:

```
# $ pip3 install pygame
import pygame as pg

pg.init()
screen = pg.display.set_mode((500, 500))
rect = pg.Rect(240, 240, 20, 20)
while not pg.event.get(pg.QUIT):
    deltas = {pg.K_UP: (0, -20), pg.K_RIGHT: (20, 0), pg.K_DOWN: (0, 20), pg.K_LEFT: (-20, 0)}
    for event in pg.event.get(pg.KEYDOWN):
        dx, dy = deltas.get(event.key, (0, 0))
        rect = rect.move((dx, dy))
    screen.fill(pg.Color('black'))
    pg.draw.rect(screen, pg.Color('white'), rect)
    pg.display.flip()
pg.quit()
```

Rect

Object for storing rectangular coordinates.

<Rect> = pg.Rect(x, y, width, height)	# Creates Rect object. Truncates passed floats
<int> = <Rect>.x/y/centerx/centery/...	# Top, right, bottom, left. Allows assignments
<tuple> = <Rect>.topleft/center/...	# Topright, bottomright, bottomleft. Same.
<Rect> = <Rect>.move((delta_x, delta_y))	# Use move_ip() to move in-place.
<bool> = <Rect>.collidepoint((x, y))	# Checks if rectangle contains the point.
<bool> = <Rect>.colliderect(<Rect>)	# Checks if the two rectangles overlap.
<int> = <Rect>.collidelist(<list_of_Rect>)	# Returns index of first colliding Rect or -1.
<list> = <Rect>.collidelistall(<list_of_Rect>)	# Returns indices of all colliding rectangles.

Surface

Object for representing images.

```
<Surf> = pg.display.set_mode((width, height)) # Opens new window and returns its surface.
<Surf> = pg.Surface((width, height))          # New RGB surface. RGBA if `flags=pg.SRCALPHA`
<Surf> = pg.image.load(<path/file>)           # Loads the image. Format depends on source.
<Surf> = pg.surfarray.make_surface(<np_array>) # Also `<np_arr> = surfarray.pixels3d(<Surf>)`
<Surf> = <Surf>.subsurface(<Rect>)            # Creates a new surface from the cutout.

<Surf>.fill(color)                            # Tuple, Color('#rrggbb[aa]') or Color(<name>)
<Surf>.set_at((x, y), color)                  # Updates pixel. Also <Surf>.get_at((x, y)).
<Surf>.blit(<Surf>, (x, y))                   # Draws passed surface at specified location.

from pygame.transform import scale, ...
<Surf> = scale(<Surf>, (width, height))        # Returns scaled surface.
<Surf> = rotate(<Surf>, anticlock_degrees)     # Returns rotated and scaled surface.
<Surf> = flip(<Surf>, x_bool, y_bool)          # Returns flipped surface.

from pygame.draw import line, ...
line(<Surf>, color, (x1, y1), (x2, y2), width) # Draws a line to the surface.
arc(<Surf>, color, <Rect>, from_rad, to_rad)    # Also ellipse(<Surf>, color, <Rect>, width=0)
rect(<Surf>, color, <Rect>, width=0)           # Also polygon(<Surf>, color, points, width=0)

<Font> = pg.font.Font(<path/file>, size)       # Loads TTF file. Pass None for default font.
<Surf> = <Font>.render(text, antialias, color) # Background color can be specified at the end
```

Sound

```
<Sound> = pg.mixer.Sound(<path/file/bytes>)   # WAV file or bytes/array of signed shorts.
<Sound>.play/stop()                          # Also set_volume(<float>), fadeout(msec).
```

Basic Mario Brothers Example

```
import collections, dataclasses, enum, io, itertools as it, pygame as pg, urllib.request
from random import randint

P = collections.namedtuple('P', 'x y')          # Position
D = enum.Enum('D', 'n e s w')                  # Direction
W, H, MAX_S = 50, 50, P(5, 10)                 # Width, Height, Max speed

def main():
    def get_screen():
        pg.init()
        return pg.display.set_mode((W*16, H*16))
    def get_images():
        url = 'https://gto76.github.io/python-cheatsheet/web/mario_bros.png'
        img = pg.image.load(io.BytesIO(urllib.request.urlopen(url).read()))
        return [img.subsurface(get_rect(x, 0)) for x in range(img.get_width() // 16)]
    def get_mario():
        Mario = dataclasses.make_dataclass('Mario', 'rect spd facing_left frame_cycle'.split())
        return Mario(get_rect(1, 1), P(0, 0), False, it.cycle(range(3)))
    def get_tiles():
        border = [(x, y) for x in range(W) for y in range(H) if x in [0, W-1] or y in [0, H-1]]
        platforms = [(randint(1, W-2), randint(2, H-2)) for _ in range(W*H // 10)]
        return [get_rect(x, y) for x, y in border + platforms]
    def get_rect(x, y):
        return pg.Rect(x*16, y*16, 16, 16)
    run(get_screen(), get_images(), get_mario(), get_tiles())

def run(screen, images, mario, tiles):
    clock = pg.time.Clock()
    pressed = set()
    while not pg.event.get(pg.QUIT):
        clock.tick(28)
        pressed |= {e.key for e in pg.event.get(pg.KEYDOWN)}
        pressed -= {e.key for e in pg.event.get(pg.KEYUP)}
        update_speed(mario, tiles, pressed)
        update_position(mario, tiles)
        draw(screen, images, mario, tiles)

def update_speed(mario, tiles, pressed):
    x, y = mario.spd
    x += 2 * ((pg.K_RIGHT in pressed) - (pg.K_LEFT in pressed))
    x += (x < 0) - (x > 0)
    y += 1 if D.s not in get_boundaries(mario.rect, tiles) else (pg.K_UP in pressed) * -10
    mario.spd = P(x=max(-MAX_S.x, min(MAX_S.x, x)), y=max(-MAX_S.y, min(MAX_S.y, y)))

def update_position(mario, tiles):
    x, y = mario.rect.topleft
    n_steps = max(abs(s) for s in mario.spd)
    for _ in range(n_steps):
        mario.spd = stop_on_collision(mario.spd, get_boundaries(mario.rect, tiles))
        x, y = x + (mario.spd.x / n_steps), y + (mario.spd.y / n_steps)
    mario.rect.topleft = x, y

def get_boundaries(rect, tiles):
    deltas = {D.n: P(0, -1), D.e: P(1, 0), D.s: P(0, 1), D.w: P(-1, 0)}
    return {d for d, delta in deltas.items() if rect.move(delta).collidelist(tiles) != -1}

def stop_on_collision(spd, bounds):
    return P(x=0 if (D.w in bounds and spd.x < 0) or (D.e in bounds and spd.x > 0) else spd.x,
            y=0 if (D.n in bounds and spd.y < 0) or (D.s in bounds and spd.y > 0) else spd.y)

def draw(screen, images, mario, tiles):
    screen.fill((85, 168, 255))
    mario.facing_left = mario.spd.x < 0 if mario.spd.x else mario.facing_left
    is_airborne = D.s not in get_boundaries(mario.rect, tiles)
    image_index = 4 if is_airborne else (next(mario.frame_cycle) if mario.spd.x else 6)
    screen.blit(images[image_index + (mario.facing_left * 9)], mario.rect)
    for t in tiles:
        is_border = t.x in [0, (W-1)*16] or t.y in [0, (H-1)*16]
        screen.blit(images[18 if is_border else 19], t)
    pg.display.flip()

if __name__ == '__main__':
    main()
```


Pandas

Data analysis library. For examples see [Plotly](#).

```
# $ pip3 install pandas matplotlib
import pandas as pd, matplotlib.pyplot as plt
```

Series

Ordered dictionary with a name.

```
>>> s = pd.Series([1, 2], index=['x', 'y'], name='a'); s
x    1
y    2
Name: a, dtype: int64
```

```
<S> = pd.Series(<list>)          # Uses list's indices for 'index'.
<S> = pd.Series(<dict>)         # Uses dictionary's keys for 'index'.
```

```
<el> = <S>.loc[key]              # Or: <S>.iloc[i]
<S> = <S>.loc[coll_of_keys]      # Or: <S>.iloc[coll_of_i]
<S> = <S>.loc[from_key : to_key_inc] # Or: <S>.iloc[from_i : to_i_exc]
```

```
<el> = <S>[key/i]                # Or: <S>.<key>
<S> = <S>[coll_of_keys/coll_of_i] # Or: <S>[key/i : key/i]
<S> = <S>[<S_of_bools>]          # Or: <S>.loc/iloc[<S_of_bools>]
```

```
<S> = <S> > <el/S>              # Returns S of bools. For logic use &, |, ~.
<S> = <S> + <el/S>              # Items with non-matching keys get value NaN.
```

```
<S> = <S>.head/describe/sort_values() # Also <S>.unique/value_counts/round/dropna().
<S> = <S>.str.strip/lower/contains/replace() # Also split().str[i] or split(expand=True).
<S> = <S>.dt.year/month/day/hour        # Use pd.to_datetime(<S>) to get S of datetimes
<S> = <S>.dt.to_period('y/m/d/h')      # Quantizes datetimes into Period objects.
```

```
<S>.plot.line/area/bar/pie/hist()      # Generates a plot. `plt.show()` displays it.
```

También '**<S>.quantile(<float/coll>)**' y '**pd.cut(<S>, bins=<int/coll>)**'.

Los objetos de indexación no pueden ser tuplas porque '**obj[x, y]**' se convierten a '**obj[(x, y)]**'.

Pandas utiliza tipos NumPy como '**np.int64**'. La serie se convierte en '**float64**' si asignamos **np.nan** a cualquier elemento. Use '**<S>.astype(<str/type>)**' para obtener la serie convertida.

¡La serie se desbordará silenciosamente si corremos '**pd.Series([100], dtype="int8") + 100**'!

Serie – Agregado, Transformación, Mapa:

```
<el> = <S>.sum/max/mean/idxmax/all/count() # Or: <S>.agg(lambda <S>: <el>)
<S> = <S>.rank/diff/cumsum/ffill/interpol...() # Or: <S>.agg/transform(lambda <S>: <S>)
<S> = <S>.isna/fillna/isin([<el/coll>])      # Or: <S>.agg/transform/map(lambda <el>: <el>)
```

	'sum'	['sum']	{ 's': 'sum' }
s.apply(...) s.agg(...)	3	sum 3	s 3

	'rank'	['rank']	{ 'r': 'rank' }
s.apply(...) s.agg(...)	x 1.0 y 2.0	x rank 1.0 y 2.0	r x 1.0 y 2.0

El último resultado tiene un índice múltiple. Utilízelo '`<S>[key_1, key_2]`' para obtener sus valores.

Marco de datos

Tabla con filas y columnas etiquetadas.

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], index=['a', 'b'], columns=['x', 'y']); df
```

	x	y
a	1	2
b	3	4

<code><DF></code>	<code>= pd.DataFrame(<list_of_rows>)</code>	# Rows can be either lists, dicts or series.
<code><DF></code>	<code>= pd.DataFrame(<dict_of_columns>)</code>	# Columns can be either lists, dicts or series.
<code><el></code>	<code>= <DF>.loc[row_key, col_key]</code>	# Or: <code><DF>.iloc[row_i, col_i]</code>
<code><S/DF></code>	<code>= <DF>.loc[row_key/s]</code>	# Or: <code><DF>.iloc[row_i/s]</code>
<code><S/DF></code>	<code>= <DF>.loc[:, col_key/s]</code>	# Or: <code><DF>.iloc[:, col_i/s]</code>
<code><DF></code>	<code>= <DF>.loc[row_bools, col_bools]</code>	# Or: <code><DF>.iloc[row_bools, col_bools]</code>
<code><S/DF></code>	<code>= <DF>[col_key/s]</code>	# Or: <code><DF>.<col_key></code>
<code><DF></code>	<code>= <DF>[<S_of_bools>]</code>	# Filters rows. For example <code>`df[df.x > 1]`</code> .
<code><DF></code>	<code>= <DF>[<DF_of_bools>]</code>	# Assigns NaN to items that are False in bools.
<code><DF></code>	<code>= <DF> > <el/S/DF></code>	# Returns DF of bools. Treats series as a row.
<code><DF></code>	<code>= <DF> + <el/S/DF></code>	# Items with non-matching keys get value NaN.
<code><DF></code>	<code>= <DF>.set_index(col_key)</code>	# Replaces row keys with column's values.
<code><DF></code>	<code>= <DF>.reset_index(drop=False)</code>	# Drops or moves row keys to column named index
<code><DF></code>	<code>= <DF>.sort_index(ascending=True)</code>	# Sorts rows by row keys. Use <code>`axis=1`</code> for cols
<code><DF></code>	<code>= <DF>.sort_values(col_key/s)</code>	# Sorts rows by passed column/s. Also <code>`axis=1`</code> .
<code><DF></code>	<code>= <DF>.head/tail/sample(<int>)</code>	# Returns first, last, or random n rows.
<code><DF></code>	<code>= <DF>.describe()</code>	# Describes columns. Also <code>info()</code> , <code>corr()</code> , <code>shape</code>
<code><DF></code>	<code>= <DF>.query('<query>')</code>	# Filters rows. For example <code>`df.query('x > 1')`</code>
<code><DF>.plot.line/area/bar/scatter(x=col_key, ...)</code>		# <code>`y=col_key/s`</code> . Also <code>hist/box(column/by=col_k)</code>
<code>plt.show()</code>		# Displays the plot. Also <code>plt.savefig(<path>)</code> .

DataFrame – Fusión, unión, concatenación:

```
>>> df_2 = pd.DataFrame([[4, 5], [6, 7]], index=['b', 'c'], columns=['y', 'z']); df_2
```

	y	z
b	4	5
c	6	7

	'outer'	'inner'	'left'	Description
df.merge(df_2, on='y', how=...)	x y z 0 1 2 . 1 3 4 5 2 . 6 7	x y z 3 4 5	x y z 1 2 . 3 4 5	Merges on column if 'on' or 'left_on/right_on' are set, else on shared cols. Uses 'inner' by default.
df.join(df_2, lsuffix='l', rsuffix='r', how=...)	x yl yr z a 1 2 . . b 3 4 4 5 c . . 6 7	x yl yr z 3 4 4 5	x yl yr z 1 2 . . 3 4 4 5	Merges on row keys. Uses 'left' by default. If Series is passed, it is treated as a column.
pd.concat([df, df_2], axis=0, join=...)	x y z a 1 2 . b 3 4 . b . 4 5 c . 6 7	y 2 4 4 6		Adds rows at the bottom. Uses 'outer' by default. A Series is treated as a column. To add a row use pd.concat([df, DF([s])]).
pd.concat([df, df_2], axis=1, join=...)	x y y z a 1 2 . . b 3 4 4 5 c . . 6 7	x y y z 3 4 4 5		Adds columns at the right end. Uses 'outer' by default. A Series is treated as a column.

DataFrame: agregar, transformar, mapear:

```
<S> = <DF>.sum/max/mean/idxmax/all/count() # Or: <DF>.apply/agg(lambda <S>: <el>)
<DF> = <DF>.rank/diff/cumsum/ffill/interpo...() # Or: <DF>.apply/agg/transform(lambda <S>: <S>)
<DF> = <DF>.isna/fillna/isin([<el/coll>]) # Or: <DF>.applymap(lambda <el>: <el>)
```

	'sum'	['sum']	{'x': 'sum'}
df.apply(...) df.agg(...)	x 4 y 6	x y sum 4 6	x 4

	'rank'	['rank']	{'x': 'rank'}
df.apply(...) df.agg(...) df.transform(...)	x y a 1.0 1.0 b 2.0 2.0	x y rank rank a 1.0 1.0 b 2.0 2.0	x a 1.0 b 2.0

Los métodos enumerados procesan las columnas a menos que reciban '**axis=1**'. Las excepciones a esta regla son '**<DF>.dropna()**', '**<DF>.drop(row_key/s)**' y '**<DF>.rename(<dict/func>)**'.

Las columnas del quinto resultado están indexadas con un índice múltiple. Esto significa que necesitamos una tupla de claves de columna para especificar una columna: '**<DF>.loc[row_key, (col_key_1, col_key_2)]**'.

Multi-índice

```
<DF> = <DF>.loc[row_key_1] # Or: <DF>.xs(row_key_1)
<DF> = <DF>.loc[:, (slice(None), col_key_2)] # Or: <DF>.xs(col_key_2, axis=1, level=1)
<DF> = <DF>.set_index(col_keys) # Creates index from cols. Also `append=False`.
<DF> = <DF>.pivot_table(index=col_key/s) # `columns=key/s, values=key/s, aggfunc='mean'`
<S> = <DF>.stack/unstack(level=-1) # Combines col keys with row keys or vice versa
```

Formatos de archivos

```
<S/DF> = pd.read_json/pickle(<path/url/file>) # Also io.StringIO(<str>), io.BytesIO(<bytes>).
<DF> = pd.read_csv/excel(<path/url/file>) # Also `header/index_col/dtype/usecols/...=<obj>`
<list> = pd.read_html(<path/url/file>) # Raises ImportError if webpage has zero tables
<S/DF> = pd.read_parquet/feather/hdf(<path...>) # Read_hdf() accepts `key=<s/df_name>` argument
<DF> = pd.read_sql('<table/query>', <conn>) # Pass SQLite3/Alchemy connection (see #SQLite)
```

```

<DF>.to_json/csv/html/parquet/latex(<path>) # Returns a string/bytes if path is omitted.
<DF>.to_pickle/excel/feather/hdf(<path>) # To_hdf() requires `key=<s/df_name>` argument.
<DF>.to_sql('<table_name>', <connection>) # Also `if_exists='fail/replace/append'`.

```

'\$ pip3 install "pandas[excel]" odfpy lxml pyarrow' instala dependencias.

`Read_csv()` solo analiza las fechas de las columnas que se especificaron con el argumento `'parse_dates'`. Intenta detectar automáticamente el formato, pero se puede ayudar con los argumentos `'date_format'` o `'dayfirst'`. Tanto las fechas como las horas se almacenan como objetos `pd.Timestamp`.

Si `'parse_dates'` e `'index_col'` son la misma columna, obtenemos un DF con `DatetimeIndex`. Su **'resample("y/m/d/h")'** método devuelve un objeto `Resampler` que es similar a `GroupBy`.

Agrupar por

Objeto que agrupa filas de un marco de datos en función del valor de la columna pasada.

```

<GB> = <DF>.groupby(col_key/s) # Splits DF into groups based on passed column.
<DF> = <GB>.apply/filter(<func>) # Filter drops a group if func returns False.
<DF> = <GB>.get_group(<el>) # Selects a group by grouping column's value.
<S> = <GB>.size() # S of group sizes. Same keys as get_group().
<GB> = <GB>[col_key] # Single column GB. All operations return S.

<DF> = <GB>.sum/max/mean/idxmax/all() # Or: <GB>.agg(lambda <S>: <el>)
<DF> = <GB>.rank/diff/cumsum/ffill() # Or: <GB>.transform(lambda <S>: <S>)
<DF> = <GB>.fillna(<el>) # Or: <GB>.transform(lambda <S>: <S>)

```

Divide las filas en grupos y suma sus columnas. El resultado tiene un índice con nombre que crea una columna **'z'** en `reset_index()`:

```

>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 6]], list('abc'), list('xyz'))
>>> gb = df.groupby('z'); gb.apply(print)
  x  y  z
a  1  2  3
  x  y  z
b  4  5  6
c  7  8  6
>>> gb.sum()
  x  y
z
3   1  2
6  11 13

```

Laminación

Objeto para cálculos de ventanas móviles.

```

<RS/RDF/RGB> = <S/DF/GB>.rolling(win_size) # Also: `min_periods=None, center=False`.
<RS/RDF/RGB> = <RDF/RGB>[col_key/s] # Or: <RDF/RGB>.<col_key>
<S/DF> = <R>.mean/sum/max() # Or: <R>.apply/agg(<agg_func/str>)

```

Trama

```

# $ pip3 install plotly kaleido pandas
import plotly.express as px, pandas as pd

```

```

<Fig> = px.line(<DF>, x=col_key, y=col_key) # Or: px.line(x=<list>, y=<list>)
<Fig>.update_layout(margin=dict(t=0, r=0, b=0, l=0)) # Also `paper_bgcolor='rgb(0, 0, 0)`.
<Fig>.write_html/json/image('<path>') # <Fig>.show() displays the plot.

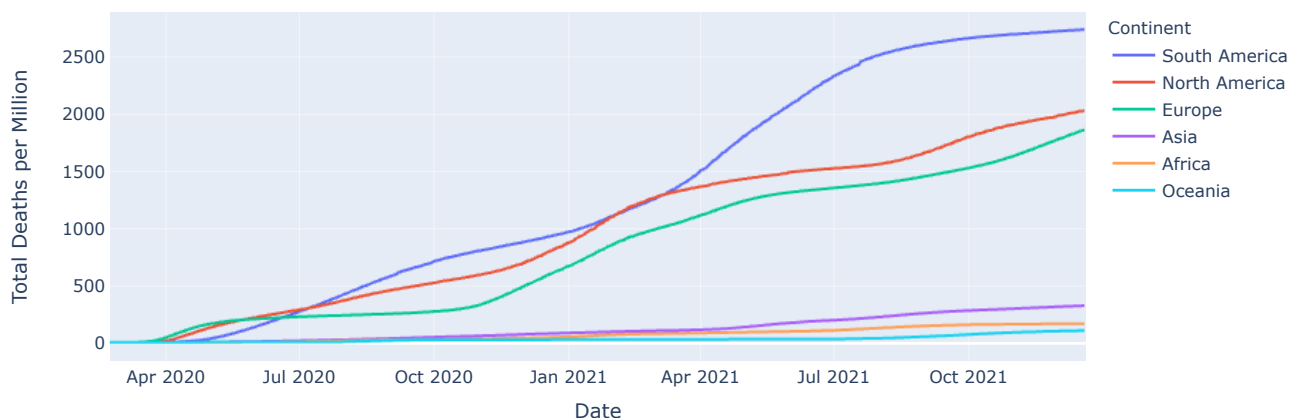
```

```

<Fig> = px.area/bar/box(<DF>, x=col_key, y=col_key)      # Also `color=col_key`.
<Fig> = px.scatter(<DF>, x=col_key, y=col_key)           # Also `color/size/symbol=col_key`.
<Fig> = px.scatter_3d(<DF>, x=col_key, y=col_key, ...)   # `z=col_key`. Also color/size/symbol.
<Fig> = px.histogram(<DF>, x=col_key [, nbins=<int>])    # Number of bins depends on DF size.

```

Muestra un gráfico lineal del total de muertes por coronavirus por millón agrupadas por continente:

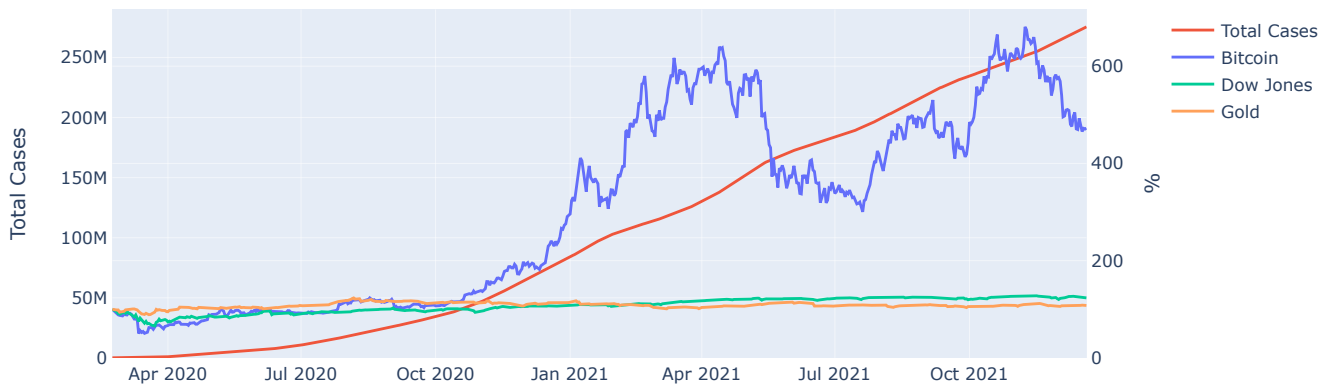


```

covid = pd.read_csv('https://raw.githubusercontent.com/owid/covid-19-data/8dde8ca49b'
                    '6e648c17dd420b2726ca0779402651/public/data/owid-covid-data.csv',
                    usecols=['iso_code', 'date', 'total_deaths', 'population'])
continents = pd.read_csv('https://gto76.github.io/python-cheatsheet/web/continents.csv',
                          usecols=['Three_Letter_Country_Code', 'Continent_Name'])
df = pd.merge(covid, continents, left_on='iso_code', right_on='Three_Letter_Country_Code')
df = df.groupby(['Continent_Name', 'date']).sum().reset_index()
df['Total Deaths per Million'] = df.total_deaths * 1e6 / df.population
df = df[df.date > '2020-03-14']
df = df.rename({'date': 'Date', 'Continent_Name': 'Continent'}, axis='columns')
px.line(df, x='Date', y='Total Deaths per Million', color='Continent').show()

```

Muestra un gráfico de líneas de varios ejes del total de casos de coronavirus y los cambios en los precios de Bitcoin, Dow Jones y el oro:



```
# $ pip3 install pandas lxml selenium plotly
import pandas as pd, selenium.webdriver, plotly.graph_objects as go

def main():
    covid, (bitcoin, gold, dow) = get_covid_cases(), get_tickers()
    df = wrangle_data(covid, bitcoin, gold, dow)
    display_data(df)

def get_covid_cases():
    url = 'https://covid.ourworldindata.org/data/owid-covid-data.csv'
    df = pd.read_csv(url, usecols=['location', 'date', 'total_cases'], parse_dates=['date'])
    df = df[df.location == 'World']
    s = df.set_index('date').total_cases
    return s.rename('Total Cases')

def get_tickers():
    with selenium.webdriver.Chrome() as driver:
        symbols = {'Bitcoin': 'BTC-USD', 'Gold': 'GC=F', 'Dow Jones': '%5EDJI'}
        for name, symbol in symbols.items():
            yield get_ticker(driver, name, symbol)

def get_ticker(driver, name, symbol):
    url = f'https://finance.yahoo.com/quote/{symbol}/history/'
    driver.get(url + '?period1=1579651200&period2=9999999999')
    if buttons := driver.find_elements('xpath', '//button[@name="reject"]'):
        buttons[0].click()
    dataframes = pd.read_html(driver.page_source, parse_dates=['Date'])
    s = dataframes[0].set_index('Date').Open
    return s.rename(name)

def wrangle_data(covid, bitcoin, gold, dow):
    df = pd.concat([bitcoin, gold, dow], axis=1) # Creates table by joining columns on dates.
    df = df.sort_index().interpolate() # Sorts rows by date and interpolates NaN-s.
    df = df.loc['2020-02-23':'2021-12-20'] # Keeps rows between specified dates.
    df = (df / df.iloc[0]) * 100 # Calculates percentages relative to day 1.
    df = df.join(covid) # Adds column with covid cases.
    return df.sort_values(df.index[-1], axis=1) # Sorts columns by last day's value.

def display_data(df):
    figure = go.Figure()
    for col_name in reversed(df.columns):
        yaxis = 'y1' if col_name == 'Total Cases' else 'y2'
        trace = go.Scatter(x=df.index, y=df[col_name], yaxis=yaxis, name=col_name)
        figure.add_trace(trace)
    figure.update_layout(
        width=944,
        height=423,
        yaxis1=dict(title='Total Cases', rangemode='tozero'),
        yaxis2=dict(title='%', rangemode='tozero', overlaying='y', side='right'),
        colorway=['#EF53B', '#636EFA', '#00CC96', '#FFA152'],
        legend=dict(x=1.08)
    )
    figure.show()

if __name__ == '__main__':
    main()
```

Apéndice

Citón

Biblioteca que compila código similar a Python en C.

```
# $ pip3 install cython
import pyximport; pyximport.install() # Module that runs imported Cython scripts.
import <cython_script>                # Script must be saved with '.pyx' extension.
<cython_script>.main()                 # Main() isn't automatically executed.
```

Definiciones:

Todas '**cdef**' las definiciones son opcionales, pero contribuyen a la aceleración.
También admite punteros C a través de '*' y '&', estructuras, uniones y enumeraciones.

```
cdef <ctype/type> <var_name> [= <obj>]
cdef <ctype>[n_elements] <var_name> [= <coll_of_nums>]
cdef <ctype/type/void> <func_name>(<ctype/type> <arg_name>): ...
```

```
cdef class <class_name>:
    cdef public <ctype/type> <attr_name>
    def __init__(self, <ctype/type> <arg_name>):
        self.<attr_name> = <arg_name>
```

Entornos virtuales

Sistema para instalar bibliotecas directamente en el directorio del proyecto.

```
$ python3 -m venv NAME          # Creates virtual environment in current directory.
$ source NAME/bin/activate      # Activates env. On Windows run `NAME\Scripts\activate`.
$ pip3 install LIBRARY         # Installs the library into active environment.
$ python3 FILE                  # Runs the script in active environment. Also `./FILE`.
$ deactivate                    # Deactivates the active virtual environment.
```

Plantilla de guión básico

Ejecute el script con '\$ python3 FILE' o '\$ chmod u+x FILE; ./FILE'. Para iniciar automáticamente el depurador cuando se produce una excepción no detectada, ejecute '\$ python3 -m pdb -cc FILE'.

```
#!/usr/bin/env python3
#
# Usage: .py
#

from sys import argv, exit
from collections import defaultdict, namedtuple
from dataclasses import make_dataclass
from enum import Enum
import functools as ft, itertools as it, operator as op, re

def main():
    pass

###
## UTIL
#

def read_file(filename):
    with open(filename, encoding='utf-8') as file:
        return file.readlines()

if __name__ == '__main__':
    main()
```

Índice

Generalmente Ctrl+F / ⌘F es suficiente.

La búsqueda '**#<title>**' limitará la búsqueda a los títulos.

Haga clic en el título '**#**' para obtener un enlace a su sección.