July 27, 2026 / Jure Šorn

Comprehensive Python Cheatsheet

Download text file, Fork me on GitHub, Check out FAQ or Switch to dark theme.

```
ToC = {
    '1. Collections':   [List, Dictionary, Set, Tuple, Range, Enumerate, Iterator, Generator],
    '2. Data Types':    [Type, String, Regular_Exp, Format, Numbers, Combinatorics, Datetime],
    '3. Syntax Rules':  [Function, Inline, Import, Decorator, Class, Duck_Type, Enum, Except],
    '4. System Calls':  [Exit, Print, Input, Command_Line_Arguments, Open, Path, OS_Commands],
    '5. Data Formats':  [JSON, Pickle, CSV, SQLite, Bytes, Struct, Array, Memory_View, Deque],
    '6. Misc Topics':   [Operator, Match_Statement, Logging, Introspection, Threads, Asyncio],
    '7. Pip Packages':  [Progress_Bar, Plot, Table, Console_App, GUI, Scraping, Web, Profile],
    '8. Multimedia':    [NumPy, Image, Animation, Audio, Synthesizer, Pygame, Pandas, Plotly]
}
```

Main

```
if __name__ == '__main__':      # Skips indented lines of code if file was imported.
    main()                      # Executes user-defined `def main(): ...` function.
```

List

```
<list> = [<el>, <el>, ...]      # Creates new list object. E.g. `list_a = [1, 2, 3]`.
```

```
<el>   = <list>[index]          # First index is 0, last -1. Also `<list>[i] = <el>`.
<list> = <list>[<slice>]        # Also <list>[from_inclusive : to_exclusive : ±step].
```

```
<list>.append(<el>)             # Appends element to the end. Or `<list> += [<el>]`.
<list>.extend(<coll>)           # Appends collection's items. Or `<list> += <coll>`.
```

```
<list>.sort()                   # Sorts in ascending order. Accepts `reverse=True`.
<list>.reverse()                # Reverses the order of elements. Takes linear time.
<list> = sorted(<coll>)         # Returns a new sorted list. Accepts `reverse=True`.
<iter> = reversed(<list>)       # Returns reversed iterator. Also list(<iterator>).
```

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

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

- **For details about functions sort(), sorted(), max() and min() see duck type sortable (p. 16).**
- **Listed lambda expressions can be replaced by the operator's itemgetter() function (p. 31).**
- **This text uses the term *collection* instead of *iterable*. For rationale see duck types (p. 18).**

```
<int> = len(<list/dict/set/…>)  # Returns number of items. Doesn't accept iterators.
<int> = <list>.count(<el>)      # Counts occurrences. Also `if <el> in <coll>: ...`.
<int> = <list>.index(<el>)      # Returns index of first occ. or raises ValueError.
<el>  = <list>.pop()            # Removes item from the end (or at index if passed).
<list>.insert(<int>, <el>)      # Inserts item at index and shifts remaining items.
<list>.remove(<el>)             # Removes the first occurrence or raises ValueError.
<list>.clear()                  # Removes all items. Also provided by dict and set.
```

Dictionary

```
<dict> = {key: val, key: val, ...}           # Use `<dict>[key]` to get or assign the value.
```

```
<view> = <dict>.keys()                       # A collection of keys reflecting all changes.
<view> = <dict>.values()                     # A collection of values that reflects changes.
<view> = <dict>.items()                      # Coll. of tuples. Each contains key and value.
```

```
value  = <dict>.get(key, default=None)       # Returns 'default' argument if key is missing.
value  = <dict>.setdefault(key, default)     # Returns/writes 'default' when key is missing.
<dict> = collections.defaultdict(<type>)     # Dict with automatic default value `<type>()`.
```

```
<dict> = dict(<collection>)                  # Creates a dict from coll. of key-value pairs.
<dict> = dict(zip(keys, values))             # Creates key-value pairs from two collections.
<dict> = dict.fromkeys(keys [, value])       # Items get value None if only keys are passed.
```

```
<dict>.update(<dict>)                        # Adds items to dict. Passed dict has priority.
value = <dict>.pop(key)                      # Removes item or raises KeyError when missing.
{k for k, v in <dict>.items() if v == 123}   # Returns a set of keys whose value equals 123.
{k: v for k, v in <dict>.items() if k in ks}  # Returns a dict of items with specified keys.
```

Counter

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

Set

```
<set> = {<el>, <el>, ...}           # Coll. of unique items. Also set(), set(<coll>).
```

```
<set>.add(<el>)                     # Adds item to the set. Same as `<set> |= {<el>}`.
<set>.update(<coll> [, ...])        # Adds items to the set. Same as `<set> |= <set>`.
```

```
<set> = <set>.union(<coll>)         # Returns a set of all items. Also <set> | <set>.
<set> = <set>.intersection(<coll>)  # Returns every shared item. Also <set> & <set>.
<set> = <set>.difference(<coll>)    # Returns set's unique items. Also <set> - <set>.
```

```
<bool> = <set>.issuperset(<coll>)   # Returns False when collection has unique items.
<bool> = <set>.issubset(<coll>)     # Is collection a superset? Also <set> <= <set>.
```

```
<el> = <set>.pop()                  # Removes one of items. Raises KeyError if empty.
<set>.remove(<el>)                  # Removes the item or raises KeyError if missing.
<set>.discard(<el>)                 # Same as remove() but it doesn't raise an error.
```

Frozen Set

- **Frozenset is immutable and hashable version of the normal set.**
- **That means it can be used as a key in a dict or as an item in a set.**

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

Tuple

Tuple is an immutable and hashable list.

```
<tuple> = ()                      # Returns an empty tuple. Also tuple(), tuple(<coll>).
<tuple> = (<el>,)                 # Returns tuple with one element. Or `<tup.> = <el>,`.
<tuple> = (<el>, <el> [, ...])    # Returns a tuple. Or `<tuple> = <el>, <el> [, ...]`.
```

Named Tuple

Tuple's subclass with named elements.

```
>>> import collections as co
>>> Point = co.namedtuple('Point', 'x y')
>>> p = Point(1, y=2)
>>> print(p)
Point(x=1, y=2)
>>> p.x, p[1]
(1, 2)
```

Range

A sequence of evenly spaced integers.

```
<range> = range(stop)               # I.e. range(to_exclusive). Ints from 0 to `stop-1`.
<range> = range(start, stop)        # I.e. range(from, to_exc). From start to `stop-1`.
<range> = range(start, stop, step)  # I.e. range(from_inclusive, to_exclusive, ±step).
```

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

Enumerate

Iterator that zips collection with range.

```
for i, el in enumerate(<coll>):
    print(f'Element {el} has index {i}.')
```

Iterator

Potentially endless stream of elements.

```
import itertools as it
```

```
<iter> = iter(<coll>)                  # Iterator that returns passed elements one by one.
<iter> = iter(<func>, to_exc)          # Calls `<func>()` until it receives 'to_exc' value.
<iter> = (<expr> for <name> in <coll>) # E.g. `(i+1 for i in range(3))`. Evaluates lazily.
<el>   = next(<iter> [, default])      # Raises StopIteration or returns 'default' on end.
<list> = list(<iter>)                  # Returns a list of iterator's remaining elements.
```

```
<iter> = it.count(start=0, step=1)     # Returns updated 'start' endlessly. Accepts floats.
<iter> = it.repeat(<obj> [, times])    # Returns passed element endlessly or 'times' times.
<iter> = it.cycle(<coll>)              # Repeats the sequence endlessly. Accepts iterators.
```

```
<iter> = it.chain(<coll>, <coll>, ...)  # Returns each element of each collection in order.
<iter> = it.chain.from_iterable(<coll>) # Accepts collection (i.e. iterable) of collections.
<iter> = it.islice(<coll>, stop)        # Also accepts 'start' and 'step'. Args can be None.
<iter> = it.product(<coll>, <coll>)     # Same as `((a, b) for a in arg_1 for b in arg_2)`.
```

- **For loops call `'iter(<coll/iter>)'`, latter returning unmodified iterator.**

Generator

- **Any function that contains a yield statement returns a generator.**
- **Generators and iterators are interchangeable (see p. 17 for details).**

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

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

Type

- **All values in Python are objects.**
- **Every object has a certain type.**
- **Type and class are synonymous.**

```
<type> = type(<obj>)                    # Object's type. Also `<obj>.__class__`.
<bool> = isinstance(<obj>, <type>)      # Also `issubclass(type(<obj>), <type>)`.
```

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

Some types do not have built-in names, so they must be imported:

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

Abstract Base Classes

Each abstract base class specifies a set of virtual subclasses. These classes are then recognized by isinstance() and issubclass() as subclasses of the ABC, although they are really not. An ABC can also manually decide whether or not a specific class is its virtual subclass, usually based on which methods that class has implemented. For instance, Iterable ABC looks for method iter(), while Collection ABC looks for iter(), contains() and 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	✓				

String

Immutable sequence of characters.

```
<str>  = 'abc'                          # Also "abc". Interprets \n, \t, \x00-\xff, etc.
```

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

```
<list> = <str>.split()                  # Splits it on one or more whitespace characters.
<list> = <str>.split(<str>)             # Splits on passed string. Also `maxsplit=<int>`.
<list> = <str>.splitlines()             # On [\n\r\f\v\x1c-\x1e\x85\u2028\u2029] and \r\n.
<str>  = <str>.join(<coll_of_str>)      # Joins items by using the string as a separator.
```

```
<bool> = <str> in <str>                 # Returns True if string contains the substring.
<bool> = <str>.startswith(<str>)        # Pass tuple of strings to give multiple options.
<int>  = <str>.find(<str>)              # Returns start index of the first match or `-1`.
```

```
<str>  = <str>.lower()                  # Lowers the case. Also upper/capitalize/title().
<str>  = <str>.casefold()               # Lower() that converts ß/ß to ss, Σ/ς to σ, etc.
<str>  = <str>.replace(old, new)        # Removes occurrences of string old if new is ''.
<str>  = <str>.translate(table)         # Get table via str.maketrans(<chr_to_str_dict>).
```

```
<str>  = chr(<int>)                     # Converts passed integer into Unicode character.
<int>  = ord(<str>)                     # Converts passed Unicode character into integer.
```

- **Use `'unicodedata.normalize("NFC", <str>)'` on strings like `'Motörhead'` before comparing them to other strings, because `'ö'` can be stored as one or two characters.**
- **`'NFC'` converts such characters to a single character, while `'NFD'` converts them to two.**

```
<bool> = <str>.isdecimal()              # Checks all chars for [0-9]. Also [۰-۹], [٩-٠].
<bool> = <str>.isdigit()                # Checks for [²³¹…] and isdecimal(). Also [₀-₉].
<bool> = <str>.isnumeric()              # Checks for [½¼¾…] and isdigit(). Also [零〇一…].
<bool> = <str>.isalnum()                # Checks for [ABC…] and isnumeric(). Also [ªµº…].
<bool> = <str>.isprintable()            # Checks for [ !"#…], basic emojis and isalnum().
<bool> = <str>.isspace()                # Checks for [ \t\n\r\f\v\x1c\x1d\x1e\x1f\x85…].
```

Regex

Functions for regular expression matching.

```
import re
<str>   = re.sub(r'<regex>', new, text)  # Substitutes occurrences with string 'new'.
<list>  = re.findall(r'<regex>', text)   # Returns all occurrences as string objects.
<list>  = re.split(r'<regex>', text)     # Add brackets around regex to keep matches.
<Match> = re.search(r'<regex>', text)    # Returns first occ. of the pattern or None.
<Match> = re.match(r'<regex>', text)     # Only searches at the start of the 'text'.
<iter>  = re.finditer(r'<regex>', text)  # Returns all occurrences as Match objects.
```

- **Raw string literals do not interpret escape sequences, thus enabling us to use the regex-specific escape sequences that cause SyntaxWarning in normal string literals (since 3.12).**
- **Argument `'new'` can also be a function that accepts a Match object and returns a string.**
- **Argument `'flags=re.IGNORECASE'` can be used with all functions that are listed above.**
- **Argument `'flags=re.MULTILINE'` makes `'^'` and `'$'` match the start/end of each line.**
- **Argument `'flags=re.DOTALL'` makes `'.'` also accept the `'\n'` (besides all other chars).**
- **`'re.compile(r"<regex>")'` returns a Pattern object with methods sub(), findall(), etc.**

Match Object

```
<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 as strings.
<int>   = <Match>.start()               # Returns start index of the whole match.
<int>   = <Match>.end()                 # Returns the match's end index plus one.
```

Special Sequences

```
'\d' == '[0-9]'                         # Also [۰-۹…]. Matches decimal character.
'\w' == '[a-zA-Z0-9_]'                  # Also [ª²³…]. Matches alphanumeric or _.
'\s' == '[ \t\n\r\f\v]'                 # Also [\x1c-\x1f…]. Matches 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, i.e. `'\D'`, `'\W'` or `'\S'`, for negation. All non-ASCII characters are matched if ASCII flag is used in conjunction with a capital letter.**

Format

String formatting mechanisms.

```
<str> = f'{<obj>}, {<obj>}'             # Braces can also contain expressions.
<str> = '{}, {}'.format(<obj>, <obj>)   # Or '{0}, {a}'.format(<obj>, a=<obj>).
<str> = '%s, %s' % (<obj>, <obj>)       # Old and redundant formatting method.
```

Example

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

Options

```
{<obj>:<10}                             # '<obj>     '.
{<obj>:^10}                             # '  <obj>   '.
{<obj>:>10}                             # '     <obj>'.
{<obj>:.<10}                            # '<obj>.....'.
{<obj>:0}                               # '<obj>'.
```

- **Objects are converted to strings with format() function, e.g. `'format(<obj>, "<10")'`.**
- **Options can be generated dynamically via nested braces: `f'{<obj>:{<str/int>}[…]}'`.**
- **Adding `'='` to the expression prepends it to its result, e.g. `f'{1+1=}'` returns `'1+1=2'`.**
- **Adding `'!r'` to the expression first calls result's repr() method and only then format().**

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'.
```

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:

	{<number>}	{<num>:f}	{<num>:e}	{<num>:%}
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}	{<num>:.2f}	{<num>:.2e}	{<num>:.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%

- **`'{<num>:g}'` is `'{<float>:.6}'` that strips `'.0'` and has 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. Hence `'{6.5:.0f}'` becomes a `'6'`, while `'{7.5:.0f}'` an `'8'`.**
- **The last rule only effects numbers that can be represented exactly by a float (`.5`, `.25`, ...).**

Ints

```
{90:x}                                  # Converts 90 to hexadecimal number '5a'.
{90:b}                                  # Converts 90 to binary number '1011010'.
{90:c}                                  # Converts 90 to Unicode character 'Z'.
```

Numbers

```
<integer> = int(<float/str/bool>)               # A whole number. Truncates floats.
<float>   = float(<integer/str/bool>)           # 8-byte decimal. Also <fl>e±<int>.
<complex> = complex(real=0, imag=0)             # Complex number. Also <fl> ± <fl>j.
<Fract>   = fractions.Fraction(numr, denom)     # `<Fraction> = <Fraction> / <int>`.
<Decimal> = decimal.Decimal(<str/int/tup>)      # `Decimal((1, (2,), 3)) == -2000`.
```

- **`'int(<str>)'` and `'float(<str>)'` raise ValueError exception if string is malformed.**
- **Decimal objects store numbers exactly, unlike most floats where `'1.1 + 2.2 != 3.3'`.**
- **Floats can be compared with: `'math.isclose(<float>, <float>, rel_tol=1e-9)'`.**
- **Precision of decimal operations is set with: `'decimal.getcontext().prec = <int>'`.**
- **Bools can be used anywhere ints can, since bool is a subclass of int: `'True + 1 == 2'`.**

Built-in

```
<num> = abs(<num>)                              # E.g. `abs(-50) == abs(50) == 50`.
<num> = pow(<num>, <num>)                       # E.g. `pow(3, 4) == 3 ** 4 == 81`.
<num> = round(<num> [, ndigits])                # E.g. `round(123.45, -1) == 120`.
<num> = min(<coll_of_nums>)                     # Also `max(<num>, <num> [, ...])`.
<num> = sum(<coll_of_nums>)                     # Also `math.prod(<coll_of_nums>)`.
```

Math

```
import math as mt
<num> = mt.pi/inf/nan                           # `inf*0` and `nan+1` return `nan`.
<num> = mt.sqrt/factorial(<num>)                # `sqrt(-1)` will raise ValueError.
<num> = mt.sin/cos/tan(<num>)                   # Also degrees, radians, asin, etc.
<num> = mt.log/log10/log2(<num>)                # Log() can accept 'base' argument.
```

Statistics

```
import statistics as st
<obj>  = st.mean/median(<coll>)              # Mode returns most common element.
<num>  = st.variance/stdev(<coll>)           # Estimates values from the sample.
<list> = st.quantiles(<coll>, n=4)           # Estimates cut points from sample.
```

Random

```
import random as rd
<num> = rd.random()                          # Selects random float from [0, 1).
<num> = rd.randint/uniform(a, b)             # Selects an int/float from [a, b].
<num> = rd.gauss(mean, stdev)                # Also triangular(low, high, mode).
<obj> = rd.choice(<sequence>)                # Doesn't mutate. Also sample(p, n).
rd.shuffle(<list>)                           # Works with all mutable sequences.
```

Hex, Bin

```
<int> = 0x<hex>                              # E.g. `0xFf == 255`. Also 0b<bin>.
<int> = int('±<hex>', 16)                    # Also int('±0x<hex>/±0b<bin>', 0).
<str> = hex(<int>)                           # Returns '[-]0x<hex>'. Also bin().
```

Bitwise

```
<int> = <int> & <int>                        # E.g. `0b1100 & 0b1010 == 0b1000`.
<int> = <int> | <int>                        # E.g. `0b1100 | 0b1010 == 0b1110`.
<int> = <int> ^ <int>                        # E.g. `0b1100 ^ 0b1010 == 0b0110`.
<int> = <int> << n_bits                      # E.g. `0b1111 << 4 == 0b11110000`.
<int> = ~<int>                               # E.g. `~100 == -(100+1) == -101`.
```

Combinatorics

```
import itertools as it
```

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

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

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

Datetime

Module that provides date, time and datetime objects.

```
# $ pip3 install python-dateutil
from datetime import *
import zoneinfo, dateutil.tz
```

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

- **Times and datetimes that have defined timezone are called *aware* and ones that don't, *naive*. If time or datetime 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 (usually for one hour).**
- **Timedelta normalizes arguments to ±days, seconds (< 86 400) and microseconds (< 1M). Its str() method returns `'[±D, ]H:MM:SS[.…]'` and total_seconds() a float of 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                     # Coordinated universal time. London without DST.
<tzinfo> = timezone(<timedelta>)            # Timezone with fixed offset from universal time.
<tzinfo> = dateutil.tz.tzlocal()            # Local timezone with dynamic offset from the UTC.
<tzinfo> = zoneinfo.ZoneInfo('<iana_key>')  # 'Continent/City_Name' zone with dynamic offset.
<DTa>    = <DT>.astimezone(<tzinfo>)        # Converts to the passed or local fixed timezone.
<Ta/DTa> = <T/DT>.replace(tzinfo=<tzinfo>)  # Changes the timezone object 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 the ISO string. Raises ValueError.
<DT>     = DT.strptime(<str>, '<format>')   # Naive or aware datetime from the custom string.
<D/DTn>  = D/DT.fromordinal(<int>)          # Date or DT from days since the Gregorian NYE 1.
<DTn>    = DT.fromtimestamp(<float>)        # A local naive DT from seconds since the epoch.
<DTa>    = DT.fromtimestamp(<float>, <tz>)  # An 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>')    # Returns custom string representation of object.
<int>    = <D/DT>.toordinal()               # Days since NYE 1, ignoring DT's time and zone.
<float>  = <DTn>.timestamp()                # Seconds since the epoch from a local naive DT.
<float>  = <DTa>.timestamp()                # Seconds since the epoch from an aware datetime.
```

Format

```
>>> dta = datetime.strptime('2025-08-14 23:39:00.00 +0200', '%Y-%m-%d %H:%M:%S.%f %z')
>>> dta.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 object is naive.**
- **`'%Z'` accepts `'UTC'`, `'GMT'` or local timezone's code and returns timezone's name, `'UTC[±HH:MM]'` if timezone is nameless, or an empty string if object is naive.**

Arithmetics

```
<bool>  =  <D/DTn>  >  <D/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 the tzinfo object.
<D/DT>  =  <D/DT>   ±  <TD>                 # Returned datetime can fall into a missing hour.
<TD>    =  <TD>     ±  <TD>                 # Also `<TD> = abs(<TD>)`, `<num> = <TD> / <TD>`.
```

Function

Independent block of code that returns a value when called.

```
def my_func(<nondefault_args>): ...                 # E.g. `my_func(x, y):`.
def my_func(<default_args>): ...                    # E.g. `my_func(x=0, y=0):`.
def my_func(<nondef_args>, <def_args>): ...         # E.g. `my_func(x, y=0):`.
```

- **Function returns None if it doesn't encounter the `'return <object/expr>'` statement.**
- **Run `'global <var_name>'` inside the function before assigning to the global variable.**
- **Value of a default argument is evaluated when function is first encountered in the scope.**
- **Any mutation of a default argument value will persist between function invocations!**

Function Call

```
<obj> = <func>(<positional_args>)                   # E.g. `my_func(0, 0)`.
<obj> = <func>(<keyword_args>)                      # E.g. `my_func(x=0, y=0)`.
<obj> = <func>(<pos_args>, <key_args>)              # E.g. `my_func(0, y=0)`.
```

Splat

Splat operator, i.e. `'*'`, expands collection into positional arguments, while splatty-splat, i.e. `'**'`, expands a dictionary into keyword arguments.

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

Is the same as:

```
func(1, 2, z=3)
```

Inside Function Def

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

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

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

	`fn(x=1, y=2)`	`fn(1, y=2)`	`fn(1, 2)`
`fn(x, *args, **kwargs):`	✓	✓	✓
`fn(*args, y, **kwargs):`	✓	✓	
`fn(*, x, **kwargs):`	✓		

Collection Unpacking

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

Inside Coll Literals

```
<list>  = [*<coll> [, ...]]         # Same as `list(<coll>) [+ ...]`.
<tuple> = (*<coll>, [...])          # Same as `tuple(<coll>) [+ ...]`.
<set>   = {*<coll> [, ...]}         # Same as `set(<coll>) [| ...]`.
<dict>  = {**<dict> [, ...]}        # Last dict has priority. Also |.
```

Inline

Lambda

```
<func> = lambda: <return_val>                    # A single statement function.
<func> = lambda <arg> [, ...]: <return_val>      # Also allows default arguments.
```

Comprehensions

```
<list> = [i+1 for i in range(5)]                 # Returns `[1, 2, 3, 4, 5]`.
<iter> = (i for i in range(10) if i > 5)         # Returns `iter([6, 7, 8, 9])`.
<set>  = {i+5 for i in range(5)}                 # Returns `{5, 6, 7, 8, 9}`.
<dict> = {i: i**2 for i in range(1, 4)}          # Returns `{1: 1, 2: 4, 3: 9}`.
```

```
>>> [l+r for l in 'abc' for r in '123']          # Inner loop is on right side.
['a1', 'a2', 'a3', ..., 'c3']
```

Map, Filter, Reduce

```
from functools import reduce
```

```
<iter> = map(lambda x: x + 1, range(5))          # Returns `iter([1, 2, 3, 4, 5])`.
<iter> = filter(lambda x: x > 5, range(10))      # Returns `iter([6, 7, 8, 9])`.
<obj>  = reduce(lambda out, x: out+x, range(5))  # Returns 10. Accepts 'initial'.
```

Any, All

```
<bool> = any(<collection>)                       # Is bool(<el>) True for any el?
<bool> = all(<collection>)                       # Is it True for all (or empty)?
```

Conditional Exp

```
<obj> = <exp> if <condition> else <exp>          # Evaluates only one expression.
```

```
>>> [i if i else 'zero' for i in (0, 1, 2)]     # `any(['', [], None])` is False.
['zero', 1, 2]
```

And, Or

```
<obj> = <exp> and <exp> [and ...]                # Returns first false or last obj.
<obj> = <exp> or <exp> [or ...]                  # Returns first true or last obj.
```

Walrus Operator

```
>>> [i for ch in '0123' if (i := int(ch))]      # Assigns to var in mid-sentence.
[1, 2, 3]
```

Named Tuple, Enum, Dataclass

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

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

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

Import

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

```
import <module>                     # Imports a built-in module or `<module>.py`.
import <package>                    # Built-in package or `<package>/__init__.py`.
import <package>.<module>           # Package's module or `<package>/<module>.py`.
from <pkg/mod>[.…] import <obj>     # Imports a module, class, func or variable.
```

- **Package is a collection of modules, but it can also define its own functions, variables, etc. On a filesystem this corresponds to a directory of Python files with an optional init script.**
- **`'import <package>'` only exposes modules that are imported inside `'__init__.py'`.**
- **Directory of the file that is passed to python command serves as the root of local imports.**
- **Use relative imports, i.e. `'from .[…][<pkg/mod>[.…]] import <obj>'`, if project has scattered entry points. Another option is to install the whole project by moving its code into 'src' dir, adding 'pyproject.toml' to its root, and running `'$ pip3 install -e .'`.**

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 (any value that is referenced from within multiple nested functions gets shared).

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

```
>>> mul_by_3 = get_multiplier(3)
>>> mul_by_3(10)
30
```

Partial

Partial transforms a function by storing some (or all) of its arguments. It is useful when a function needs to be passed as an argument, e.g. `'collections.defaultdict(<func>)'`, `'iter(<func>, to_exc)'` and `'dataclasses.field(default_factory=<func>)'`.

```
def mul(a, b):
    return a * b
```

```
>>> import functools as ft
>>> mul_by_3 = ft.partial(mul, 3)
>>> mul_by_3(10)
30
```

Non-Local

If variable is being assigned to anywhere in the scope (i.e., body of a function), it is treated as a local variable unless it is declared `'global'` or `'nonlocal'` before its first usage.

```
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 (p. 17), but is usually implemented as a function that returns a closure (p. 12).

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

Debugger

Prints function's name every time function is called. It uses `'@wraps'` decorator to move the metadata from func() into out(). Without it, `'add.__name__'` would return `'out'`.

```
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
```

Cache

Stores function's return values and reuses them later. To clear stored return values run `'<func>.cache_clear()'`, or use `'@lru_cache(maxsize=<int>)'` decorator instead.

```
from functools import cache

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

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

Debug with Args

Decorator that prints function's name and optionally also it's result.

```
from functools import wraps

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

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

- Using `'@debug'` without arguments won't work here because add() is then passed via 'print_result' argument. To fix this issue use `'def debug(fn=None, *, ...)'` in def and `'return decorator(fn) if fn else decorator'` as the last line.

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)')
```

- **Methods whose names start and end with two underscores are called special methods.**
- **They are executed when object is passed to a built-in function or used as an operand. For example, `'print(a)'` calls `'a.__str__()'` and `'a + b'` calls `'a.__add__(b)'`.**
- **See Operator (p. 30) to get names of all special methods that are called by operators.**
- **Methods that are decorated with `'@staticmethod'` receive neither 'self' nor 'cls' arg.**
- **Return value of str() special method should be readable and of repr() unambiguous. All calls to str() special method are dispatched to repr() when only repr() is provided.**

Expressions that call str() special method:

```
f'{obj}'
str(obj)
print(obj)
```

Expressions that call repr() special method:

```
f'{obj!r}'
str/repr/print([obj])
str/repr/print({obj: obj})
str/repr/print(MyDataClass(obj))
```

Subclass

- **Inheritance is a mechanism that enables a class to extend some other class (i.e. sub-class to extend its parent) and by doing so inherit all of its methods and attributes.**
- **Subclass can then add its own methods and attributes or override inherited ones by reusing their names.**

```
class Person:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return f'Person({self.name!r})'
    def __lt__(self, other):
        return self.name < other.name

class Employee(Person):
    def __init__(self, name, staff_num):
        super().__init__(name)
        self.staff_num = staff_num
    def __repr__(self):
        return f'Employee({self.name!r}, {self.staff_num})'
```

```
>>> people = [Person('Bob'), Employee('Ann', 0)]
>>> sorted(people)
[Employee('Ann', 0), Person('Bob')]
```

Type Annotations

They are used by type checkers like mypy and Pydantic, however they are not enforced by CPython interpreter. To annotate a function use `'def f(a: int = 0) -> int: ...'`.

```
from collections.abc import *

<name>: <type> [| ...] [= <obj>]
<name>: list/set/Iterable/Sequence[<type>] [= <obj>]
<name>: tuple/dict[<type>, ...] [= <obj>]
```

Dataclass

It uses class variables to generate init(), repr() and eq() special methods.

```
import dataclasses as dc
```

```
@dc.dataclass(order=False, frozen=False)
class MyClass:
    <attr_name>: <type>
    <attr_name>: <type> = <obj>
    <attr_name>: list = dc.field(default_factory=list)
```

- **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 accepts any callable object.**
- **For attributes and arguments of arbitrary type use `'<attr_name>: typing.Any'`.**

Inline:

```
P = dc.make_dataclass('P', ['x', 'y'])
P = dc.make_dataclass('P', [('x', float), ('y', float)])
P = dc.make_dataclass('P', [('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 restricting objects to listed attributes.

```
class Point:
    __slots__ = ('x', 'y')
```

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 possesses all of the duck type's prescribed methods 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 (because id() returns object's memory address that is guaranteed to be unique).**
- **Only the left side object has eq() method called, unless it returns NotImplemented, in which case the right object is consulted. Result is False if both return NotImplemented.**
- **Method ne() (called by `'!='`) 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 hash() and eq() methods and its hash value must 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 the eq() method.**

```
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 (called by <, >, <=, >=) and the rest will be automatically generated.**
- **Built-in functions sorted() and min() only require lt() method, while max() only requires gt(). However, it's best to define them all so that confusion doesn't arise in other context.**
- **When two lists, strings, or data classes are compared, their values get compared one by one 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 their 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 special methods next() and iter() is an iterator.
- Next() should return the next item or raise StopIteration exception.
- Iter() should return an unmodified iterator, i.e. the 'self' argument.
- Any object that has iter() special method can be used in a for loop.

```
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'.
- Objects returned by the itertools module, such as count, repeat and cycle.
- Generator objects returned by the generator functions and expressions.
- File objects returned by the open() function, SQLite cursor objects, etc.

Callable

- All functions and classes have a call() method that is executed when they are called.
- Use `'callable(<obj>)'` or `'isinstance(<obj>, collections.abc.Callable)'` to check if object is callable and `'inspect.signature(<obj>)'` for info about args.
- When this text uses `'<function>'` as an argument, it actually means `'<callable>'`.

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

```
>>> counter = Counter()
>>> counter(1), counter(1), counter(1)
(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 (file, socket, etc.).
- Exit() should release the resources (for example close the file, release the lock, etc.).
- Any exception that happens inside the with block is passed to exit() method. Exit() can then suppress this exception by returning a true value (not None, False, 0, etc.).

```
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 special method is iter(). It should return an iterator of object's items.**
- **Special method 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 length of collection.**
- **This text refers to all iterable objects as collections, which is technically incorrect. The term *iterable* was avoided 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 actually 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 len() and getitem(). Getitem() should return an item at the passed index or raise IndexError (it may also support negative indices and/or slices).**
- **Iter() and contains() automatically work on any object with defined getitem() method.**
- **Reversed() automatically works on any object that has len() and getitem() 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() or getitem(), and sequence as any object with getitem() and len(). It doesn't define the term *collection*.**
- **Using ABC Iterable with isinstance() or issubclass() only checks whether object/class has special method iter(), while ABC Collection checks for iter(), contains() and len() (p. 4).**

ABC Sequence

- **It's a richer interface than the basic sequence that also requires just len() and getitem().**
- **Extending it generates iter(), contains(), reversed(), index() and count() special methods.**
- **Unlike `'abc.Iterable'` and `'abc.Collection'`, it is not a duck type. That is why exp. `'issubclass(MySequence, abc.Sequence)'` would return False even if MySequence had all methods defined. It however recognizes list, tuple, range, string, 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]
```

Required and automatically available methods:

	Iterable	Collection	Sequence	abc.Sequence
__iter__	!	!	✓	✓
__contains__	✓	✓	✓	✓
__len__		!	!	!
__getitem__			!	!
__reversed__			✓	✓
index				✓
count				✓

- **Method iter() is required for `'isinstance(<obj>, abc.Iterable)'` to return True, however any object with getitem() method works 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 MyEnum(Enum):
    <member_name> = auto()            # An increment of last numeric value or 1.
    <member_name> = <value>           # Values don't have to be hashable/unique.
    <member_name> = <el>, <el>, ...   # Value can be a collection, e.g. a tuple.
```

- **Methods receive the member they were called on as the 'self' argument.**
- **Accessing a member named after a reserved keyword raises SyntaxError.**

```
<memb> = <enum>.<member_name>         # Accesses a member via enum's attribute.
<memb> = <enum>['<member_name>']      # Returns the member or raises KeyError.
<memb> = <enum>(<value>)              # Returns the member or raises ValueError.
<str>  = <member>.name                # Returns the member's name as a string.
<obj>  = <member>.value               # Value can't be a user-defined function.
```

```
<list> = list(<enum>)                 # Returns a list containing every member.
<list> = <enum>._member_names_        # Returns a list containing member names.
<list> = [m.value for m in <enum>]    # Returns a list containing member values.
```

```
<enum> = type(<member>)               # Returns an enum. Also <memb>.__class__.
<iter> = itertools.cycle(<enum>)      # Returns an endless iterator of members.
<memb> = random.choice(list(<enum>))  # Randomly selects one of enum's members.
```

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:

```
import functools as ft
and_ = ft.partial(lambda l, r: l and r)
or_ = ft.partial(lambda l, r: l or r)
LogicOp = Enum('LogicOp', {'AND': and_, 'OR': or_})
```

Exceptions

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

Full Try Statement

```
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 the function block delimits scope).**
- **To catch signals use `'signal.signal(signal_number, my_handler_function)'`.**

Catching Exceptions

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

- **Except clause catches all subclasses, e.g. `'OSError'` is caught by `'except Exception:'`.**
- **Use `'traceback.print_exc()'` to print the full error message to standard error stream.**
- **Use `'print(<name>)'` to print just the cause of the exception (its arguments) to stdout.**
- **Use `'logging.exception(<str>)'` to log the passed message followed by the full error message of the caught exception. For details about the logger setup see Logging (p. 31).**
- **`'sys.exc_info()'` returns type, object and traceback of the caught exception as a tuple.**

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_str  = linecache.getline(filename, <name>.__traceback__.tb_lineno)
trace_str = ''.join(traceback.format_tb(<name>.__traceback__))
error_msg = ''.join(traceback.format_exception(*sys.exc_info()))
```

Built-in Exceptions

```
BaseException
 ├─ SystemExit                # Raised when `sys.exit()` is called. See #Exit for details.
 ├─ KeyboardInterrupt         # Raised when the user hits the interrupt key, i.e. `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 index of a sequence (list/str) is out of range.
     │   └─ KeyError          # Raised when a dictionary's key or a 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 a local name is used before it's being defined.
     ├─ OSError               # Errors such as FileExistsError and TimeoutError. See #Open.
     │   └─ ConnectionError   # Errors such as BrokenPipeError and ConnectionAbortedError.
     ├─ RuntimeError          # Is raised by errors that do not fit into other categories.
     │   ├─ NotImplementedEr… # Can be raised by abstract methods or by an unfinished code.
     │   └─ RecursionError    # Raised if max recursion depth is exceeded (3k by default).
     ├─ StopIteration         # Raised when exhausted (empty) iterator is passed to next().
     ├─ TypeError             # Raised when argument of wrong type is passed to a function.
     └─ ValueError            # Raised when it has the right type but inappropriate value.
```

Exceptions raised by collections:

	<list>	<set>	<dict>
[i/key]	IndexError		KeyError
.pop()	IndexError	KeyError	KeyError
.remove()	ValueError	KeyError	
.index()	ValueError		

Useful built-in exceptions:

```
raise TypeError('Function received argument of the wrong type!')
raise ValueError('Argument has right type but its value is off!')
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(<obj>, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

- **Use `'file=sys.stderr'` or `'sys.stderr.write(<str>)'` 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 the program exits.**

Pretty Print

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

- **Each item is printed on its own line if collection exceeds 'width' characters.**
- **Nested collections that are `'depth=<int>'` 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 stream is already exhausted.**

Arguments

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

Argument Parser

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

- **Use `'help=<str>'` to set argument description that is used by `'-h'`.**
- **Use `'default=<obj>'` to set option's or argument's default value.**

Open

Opens a file and returns the corresponding file object.

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

- **`'encoding=None'` means that a default encoding is used, which is platform dependent. Best practice is to use `'encoding="utf-8"'` until it becomes the default (Python 3.15).**
- **`'newline=None'` means that all different end of line combinations are converted to '\n' on read, while on write all '\n' characters are converted to the system's default separator.**
- **`'newline=""'` means no conversions take place, but input is still broken into chunks by readline() on every '\n', '\r' and '\r\n'. Passing `'newline="\n"'` breaks input only on '\n'.**
- **`'newline="\r\n"'` breaks input only on '\r\n' and converts every '\n' to '\r\n' on write.**

Modes

- `'r'` - Reads text from the file (the default option).
- `'w'` - Writes to the file. Deletes existing contents.
- `'x'` - Writes or raises FileExistsError if file exists.
- `'a'` - Appends. Creates new file if it doesn't exist.
- `'w+'` - Reads and writes. Deletes existing contents.
- `'r+'` - Reads and writes from the start of the file.
- `'a+'` - Reads and writes from the end of the file.
- `'rb'` - Reads bytes (p. 28). Also `'wb'`, `'xb'`, etc.

Exceptions

- `'FileNotFoundError'` can be raised when reading with `'r'` or `'r+'`.
- `'FileExistsError'` exception can be raised when writing with `'x'`.
- `'IsADirectoryError'`, `'PermissionError'` can be raised by any.
- `'except OSError [as <name>]: …'` catches all listed exceptions.

File Object

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

```
<obj>  = <file>.read(size=-1)     # Reads 'size' chars/bytes or until the EOF.
<obj>  = <file>.readline()        # Returns a line or empty string/bytes on EOF.
<list> = <file>.readlines()       # Returns remaining lines. Also list(<file>).
<obj>  = next(<file>)             # Returns a line using the read-ahead buffer.
```

```
<file>.write(<obj>)               # Writes str or bytes object to write buffer.
<file>.writelines(<coll>)         # Writes a coll. of strings or bytes objects.
<file>.flush()                    # Flushes write buff. Runs every 4096/8192 B.
<file>.close()                    # Closes a 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>, ...)  # Uses `os.sep` to join strings or Path objects.
<str>  = os.path.realpath(<path>)   # Resolves symlinks and calls os.path.abspath().
```

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

```
<list> = os.listdir(path='.')       # Returns all file/dir names located at 'path'.
<list> = glob.glob('<pattern>')     # Returns paths matching the wildcard pattern.
```

```
<bool> = os.path.exists(<path>)     # Checks if path exists. Also <Path>.exists().
<bool> = os.path.isfile(<path>)     # Also <Path>.is_file(), <DirEntry>.is_file().
<bool> = os.path.isdir(<path>)      # Also <Path>.is_dir() and <DirEntry>.is_dir().
```

```
<stat> = os.stat(<path>)            # A status object. Also <Path/DirEntry>.stat().
<num>  = <stat>.st_size/st_mtime/…  # Returns size in bytes, modification time, ...
```

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            # Is absolute if 'path' argument was absolute.
<str>  = <DirEntry>.name            # Returns the path's final component as string.
<file> = open(<DirEntry>)           # Opens the file and returns its file object.
```

Path Object

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

```
<Path> = Path()                     # Returns current working dir. Also Path('.').
<Path> = Path.cwd()                 # Returns absolute CWD. Also Path().resolve().
<Path> = Path.home()                # Returns the user's absolute home directory.
<Path> = Path(__file__)             # Use resolve() to get module's absolute path.
```

```
<Path> = <Path>.parent              # Returns the path without its final component.
<str>  = <Path>.name                # Returns final component (i.e. file/dirname).
<str>  = <Path>.suffix              # Returns the name's last extension with a dot.
<str>  = <Path>.stem                # Returns the name without its last extension.
<tup.> = <Path>.parts               # Starts with '/' or 'C:\' if path is absolute.
```

```
<iter> = <Path>.iterdir()           # Returns directory contents as Path objects.
<iter> = <Path>.glob('<patt>')      # Returns Paths matching the wildcard pattern.
```

```
<str>  = str(<Path>)                # Returns path as string. Also <Path>.as_uri().
<file> = open(<Path>)               # Also <Path>.read_text/write_bytes/…(<args>).
```

OS Commands

```
import os, shutil as sh
```

```
os.chdir(<path>)              # Changes the current working directory (or CWD).
os.mkdir(<path>)              # Creates dir. Set permissions with `mode=0o777`.
os.makedirs(<path>)           # Creates all path's dirs. Also `exist_ok=False`.
```

```
sh.copy(from, to)             # Copies file (arg. 'to' can exist or be a dir).
sh.copy2(from, to)            # Also copies the creation and modification time.
sh.copytree(from, to)         # Copies directory (arg. 'to' should not exist).
```

```
os.rename(from, to)           # Renames or moves the file or directory 'from'.
os.replace(from, to)          # Same, but overwrites file 'to' even on Windows.
sh.move(from, to)             # `rename()` that moves into 'to' if it's a dir.
```

```
os.remove(<path>)             # Deletes file. Also `$ pip3 install send2trash`.
os.rmdir(<path>)              # Deletes empty dir. Raises OSError if it's not.
sh.rmtree(<path>)             # Deletes the directory and all of its contents.
```

- **Provided paths can be either strings, Path objects, or DirEntry objects.**
- **Functions report errors by raising OSError or one of its subclasses (p. 23).**

Shell Commands

```
import os, subprocess as sp
```

```
<int>  = os.system('<cmds>')    # Runs commands in sh/cmd shell. Prints results.
<proc> = sp.run(<str/list>)     # For parameters see examples. Prints by default.
<pipe> = os.popen('<cmds>')     # Prints only stderr. Soft deprecated since 3.14.
<str>  = <pipe>.read()          # Returns combined stdout. Provides readline/s().
<int>  = <pipe>.close()         # Returns None if last command had returncode 0.
```

Sends "1+1" to the basic calculator and captures its stdout and stderr streams:

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

Sends test.in to `bc` running in standard mode and saves its stdout to test.out:

```
>>> if os.system('echo 1+1 > test.in') == 0:
...     with open('test.in') as in_, open('test.out', 'w') as out:
...         sp.run(shlex.split('bc -s'), stdin=in_, stdout=out)
...     print(open('test.out').read())
2
```

JSON

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

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

Pickle

```
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
```

```
<file> = open(<path>, newline='')              # Opens the text file for reading.
<read> = csv.reader(<file>, 'excel')           # Also `delimiter=','`. See Params.
<list> = next(<read>)                          # Returns a row as list of strings.
<list> = list(<read>)                          # Returns list of remaining rows.
```

- **For XML and binary Excel files (extensions xlsx, xlsm and xlsb) use Pandas library (p. 46).**
- **To print the spreadsheet to the console use either Tabulate or PrettyTable library (p. 34).**
- **Reader can consume any iterator or collection of strings, not just text files.**

Write

```
<file>  = open(<path>, 'a', newline='')        # Opens the text file for writing.
<write> = csv.writer(<file>, 'excel')          # Also `delimiter=','`. See Params.
<write>.writerow(<collection>)                 # Encodes objects using str(<obj>).
<write>.writerows(<coll_of_coll>)              # Appends rows to the opened file.
```

- **Always pass `'newline=""'` argument to open(), or newlines embedded inside quoted fields will flip between '\n' and '\r\n' (in some cases '\r\n' may even change to '\r\r\n'). Also rows won't be terminated with passed or dialect's `'lineterminator'` parameter.**
- **Open existing file with `'mode="a"'` to append to it or `'mode="w"'` to overwrite it.**

Params

- **`'dialect'` - Master parameter that sets the default values. String or a *csv.Dialect* object.**
- **`'delimiter'` - A one-character string that separates fields. Comma, tab, semicolon, etc.**
- **`'lineterminator'` - Sets 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. Can be None if doublequote is True.**
- **`'doublequote'` - Whether quotechars inside fields are/get doubled (instead of 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
<con> = sqlite3.connect(<path>)        # Opens existing or new file. Also ':memory:'.
<con>.close()                          # Closes connection. Discards uncommitted data.
```

Read

```
<cursor> = <con>.execute('SELECT …')   # Can raise a subclass of the `sqlite3.Error`.
<tuple>  = <cursor>.fetchone()         # Returns the next row. Same as next(<cursor>).
<list>   = <cursor>.fetchall()         # Returns remaining rows. Also list(<cursor>).
```

Write

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

Or:

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

Params

```
<con>.execute(<sql>, <list/tuple>)     # Replaces every '?' with corresponding item.
<con>.execute(<sql>, <dict/namedtup>)  # Replaces every ':<key>' with matching value.
<con>.executemany(<sql>, <colls>)      # Executes statement once for each collection.
```

- **Accepts strings, ints, floats, bytes, None objects, and bools (stored as 1 or 0).**
- **Columns are not restricted to any specific type unless table is declared strict.**

Example

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

```
>>> con = sqlite3.connect('test.db')
>>> con.execute('CREATE TABLE person (name TEXT, height INTEGER) STRICT')
>>> con.execute('INSERT INTO person VALUES (?, ?)', ('Jean-Luc', 187))
>>> con.execute('SELECT rowid, * FROM person').fetchall()
[(1, 'Jean-Luc', 187)]
```

SQLAlchemy

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

```
# $ pip3 install sqlalchemy
import sqlalchemy as sa
<eng> = sa.create_engine(<url>)        # Url: 'dialect://user:password@host/dbname'.
<con> = <eng>.connect()                # Creates new connection. Also <con>.close().
<cur> = <con>.execute(sa.text(<sql>))  # Add dict to execute() to replace ':<key>'s.
with <con>.begin(): ...                # Exits the block with a 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

An immutable sequence of single bytes. Mutable version is called bytearray.

```
<bytes> = b'<str>'                  # Accepts ASCII characters and \x00 to \xff.
<int>   = <bytes>[index]            # Returns the byte as int between 0 and 255.
<bytes> = <bytes>[<slice>]          # Returns bytes even if it has one element.
<bytes> = <bytes>.join(<coll>)      # Joins bytes objects using bytes as a sep.
```

Encode

```
<bytes> = bytes(<ints>)             # Accepts coll of integers between 0 and 255.
<bytes> = bytes(<str>, 'utf-8')     # Encodes the string. Same as <str>.encode().
<bytes> = bytes.fromhex('<hex>')    # Hex pairs can be separated by whitespaces.
<bytes> = <int>.to_bytes(n_bytes)   # Accepts `byteorder='little', signed=True`.
```

Decode

```
<list>  = list(<bytes>)             # Returns a list of ints between 0 and 255.
<str>   = str(<bytes>, 'utf-8')     # Returns a string. Same as <bytes>.decode().
<str>   = <bytes>.hex()             # Returns hex pairs separated by `sep=<str>`.
<int>   = int.from_bytes(<bytes>)   # Accepts `byteorder='little', signed=True`.
```

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

- **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>', <num>, ...)  # Packs numbers according to format.
<tuple> = unpack('<format>', <bytes>)   # Use `iter_unpack()` to get tuples.
```

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

Integers. Unsigned types use capital letters. 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 that fit into selected C 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 by calling the byteswap() method.

```
from array import array
```

```
<array> = array('<ctype>' [, <coll>])  # Creates array. Accepts collection of numbers.
<array> = array('<ctype>', <bytes>)    # Copies passed bytes into the array's memory.
<array> = array('<ctype>', <array>)    # Treats passed array as a sequence of numbers.
<array>.fromfile(<file>, n_items)      # Appends file contents to the array's memory.
```

```
<bytes> = bytes(<array>)               # Returns copy of the memory as a bytes object.
<file>.write(<array>)                  # Appends the array's memory to a 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/array>)  # Returns mutable memoryview if array is passed.
<obj>   = <mview>[index]             # Returns an int/float. Bytes if format is 'c'.
<mview> = <mview>[<slice>]           # Returns a memoryview with rearranged elements.
<mview> = <mview>.cast('<ctype>')    # Only works between B/b/c and the other types.
<mview>.release()                    # Releases the memory buffer of the base object.
```

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

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

Deque

List with efficient appends and pops from either side.

```
from collections import deque
```

```
<deque> = deque(<coll>)           # Pass `maxlen=<int>` to set the size limit.
<deque>.appendleft(<el>)          # Drops last element if maxlen is exceeded.
<deque>.extendleft(<coll>)        # Prepends reversed collection to the deque.
<deque>.rotate(n=1)               # Moves last element to the start of deque.
<el> = <deque>.popleft()          # Removes and returns deque's first element.
```

Operator

Module of functions that provide the functionality of operators. Functions are grouped by operator precedence, from least to most binding. Functions/operators in first and third line are also ordered by precedence within a line.

```
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>)             # |, ^, & (sorted by precedence).
<int>  = op.lshift/rshift(<int>, <int>)                    # <<, >> (i.e. <int> << n_bits).
<obj>  = op.add/sub(<obj>, <obj>)                          # +, - (e.g. 'a' + 'b' == 'ab').
<obj>  = op.mul/truediv/floordiv/mod(<obj>, <obj>)         # *, /, //, % (evaluated l to r).
<num>  = op.neg/invert(<num>)                              # -, ~ (negate and bitwise not).
<num>  = op.pow(<num>, <num>)                              # ** (pow() accepts 3 arguments).
<func> = op.itemgetter/attrgetter/methodcaller(<obj>, …)   # [i/key], .attr_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))
```

- **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().**
- **`'and/or'` can't be emulated by a function because they might not evaluate all operands.**
- **Comparisons can be chained: `'x < y < z'` gets converted to `'(x < y) and (y < z)'`.**

Match Statement

Executes the first block with matching pattern.

```
match <obj/expr>:
    case <pattern> [if <cond>]:
        <code>
    ...
```

Patterns

```
<val_patt> = 1/'a'/True/None/math.pi     # Matches the literal or attribute's value.
<cls_patt> = <type>()                    # Matches any object of that type (or ABC).
<wildcard> = _                           # Matches any object. Useful in last case.
<capture>  = <name>                      # Matches any object and binds it to name.
<as_patt>  = <pattern> as <name>         # Binds match to name. Also <type>(<name>).
<or_patt>  = <pattern> | ...             # Matches if any of listed patterns match.
<seq_patt> = [<pattern>, ...]            # Matches a sequence. All items must match.
<map_patt> = {<val_patt>: <patt>, ...}   # Matches a dict if it has matching items.
<cls_patt> = <type>(<name>=<patt>, ...)  # Matches object with matching attributes.
```

- **The sequence pattern can also be written as a tuple, either with or without the brackets.**
- **Use `'*<name>'` and `'**<name>'` in sequence/mapping patterns to bind remaining items.**
- **Patterns can be surrounded with brackets to override their precedence: `'|'` > `'as'` > `','`. For example, `'[1, 2]'` is matched by expression `'case 1|2, 2|3 as y if y == 2:'`.**
- **All names that are bound in the matching case, as well as variables initialized in its body, are visible after the match statement (only function block delimits scope).**

Example

```
>>> from pathlib import Path
>>> match Path('/home/bwk/documents/README.md'):
...     case Path(
...         parts=['/', 'home', user, *_, name]
...     ) as p if p.is_file() and 'readme' in name.lower():
...         print(f"{name} is {user}'s readme file.")
README.md is bwk's readme file.
```

Logging

```
import logging as log
```

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

Setup

```
log.basicConfig(
  filename=None,                                # Prints to stderr when filename is None.
  filemode='a',                                 # Use mode 'w' to overwrite existing file.
  format='%(levelname)s:%(name)s:%(message)s',  # Using '%(asctime)s' adds local datetime.
  level=log.WARNING,                            # Drops messages that have lower priority.
  handlers=[log.StreamHandler(sys.stderr)]      # Uses FileHandler when 'filename' is set.
)
```

```
<Formatr> = log.Formatter('<format>')           # Formats messages using the format str.
<Handler> = log.FileHandler(<path>, mode='a')   # Appends to file. Also `encoding=None`.
<Handler>.setFormatter(<Formatter>)             # Only outputs bare messages by default.
<Handler>.setLevel(<str/int>)                   # Prints/saves every message by default.
<Logger>.addHandler(<Handler>)                  # Loggers can have more than one handler.
<Logger>.setLevel(<str/int>)                    # What's 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>.<name>'`.**
- **Logger will inherit the level from its parent if you don't set it via the setLevel() method.**
- **Format string can contain: pathname, filename, funcName, lineno, thread and process.**
- **RotatingFileHandler rotates files according to 'maxBytes' and 'backupCount' arguments.**
- **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.**
- **Logging messages generated by libraries are passed to the root's handlers. Level of the library's logger can be set with `'log.getLogger("<library>").setLevel(<str>)'`.**

Logger that writes 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')
>>> format_str = '%(asctime)s %(levelname)s:%(name)s:%(message)s'
>>> handler.setFormatter(log.Formatter(format_str))
>>> logger.addHandler(handler)
>>> logger.setLevel('DEBUG')
>>> log.basicConfig()
>>> stream_handler = log.root.handlers[0]
>>> stream_handler.setLevel('WARNING')
>>> logger.critical('Missing config file.')
CRITICAL:my_module:Missing config file.
>>> print(open('test.log').read())
2023-02-07 23:21:01,430 CRITICAL:my_module:Missing config file.
```

Introspection

```
<list> = dir()                      # Local names of objects, incl. functions/classes.
<dict> = vars()                     # Local names and their objects. Same as locals().
<dict> = globals()                  # Global names and their objects. E.g. __builtin__.
```

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

Threading

Threads are functions that run concurrently. Things can get messy when they share objects.
`'$ uv init --python 3.15t'` installs interpreter that can run threads on multiple cores.

```
import threading as th, queue as qu
import concurrent.futures as cf
```

Thread

```
<Thread> = th.Thread(target=<func>)     # Use `args=<coll>` to set function's arguments.
<Thread>.start()                        # Runs function in background. Also is_alive().
<Thread>.join()                         # Waits until the function finishes executing.
```

- **Use `'kwargs=<dict>'` to pass keyword arguments to the function, i.e. thread.**
- **Use `'daemon=True'`, or the program won't be able to exit while thread is alive.**

Lock

```
<lock> = th.Lock/RLock()                # RLock can only be released by acquirer thread.
<lock>.acquire()                        # Waits/blocks until the lock becomes available.
<lock>.release()                        # Releases the lock so it can be acquired again.
```

Or:

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

Sync Objects

```
<Semaphr> = th.Semaphore(value=1)       # A lock that can be acquired by value threads.
<Event>   = th.Event()                  # `<Event>.wait()` blocks until set() is called.
<Barrier> = th.Barrier(parties)         # Wait() blocks until it's called parties times.
```

Queue

```
<Queue> = qu.Queue(maxsize=0)           # A first-in-first-out queue. It's thread safe.
<Queue>.put(<obj>)                      # The call blocks until queue stops being full.
<Queue>.put_nowait(<obj>)               # Raises the qu.Full exception if queue is full.
<obj> = <Queue>.get()                   # The call blocks until queue stops being empty.
<obj> = <Queue>.get_nowait()            # Raises the qu.Empty exception if it is empty.
```

Thread Executor

```
<Exec> = cf.ThreadPoolExecutor()        # Or use `with ThreadPoolExecutor() as <name>:`.
<iter> = <Exec>.map(<fn>, <args>, ...)  # Multithreaded and non-lazy map(). Keeps order.
<Futr> = <Exec>.submit(<fn>, <arg>, ...)# Queues function for execution. Returns Future.
<Exec>.shutdown()                       # Waits until all submitted tasks are completed.
```

```
<iter> = cf.as_completed(<Futrs>)       # `next(<iter>)` returns next completed Future.
<obj>  = <Future>.result()              # Raises TimeoutError if `timeout=<fl>` is used.
<bool> = <Future>.done()                # Returns True if function has finished running.
<bool> = <Future>.cancel()              # Just returns False if func is already running.
```

- **Map() and as_completed() also accept 'timeout' arg. It causes *futures.TimeoutError* when next() is called or 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 tasks are done.**
- **Exceptions that happen inside threads are raised when map's next() or Future's result() method is called. Future's exception() method returns caught exception object or None.**
- **ProcessPoolExecutor provides true parallelism but: everything sent to and 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__": ...'`.**

Asyncio

- **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 consume as much memory.**
- **Coroutine definition starts with `'async'` keyword and its call with `'await'` keyword.**
- **Execute `'asyncio.run(<coroutine>)'` to start running the first/main coroutine.**

```
import asyncio as ac
```

```
<coro> = <async_function>(<args>)        # Creates a coroutine by calling async func.
<obj>  = await <coroutine>               # Starts coroutine. Returns result or None.
<task> = ac.create_task(<coroutine>)     # Schedules coroutine. Always keep the task.
<obj>  = await <task>                    # Returns the result. Also <task>.cancel().
```

```
<coro> = ac.gather(<coro/task>, ...)     # Schedules coros. Returns list of results.
<iter> = ac.as_completed(<coros/tasks>)  # `await next(<iter>)` returns next result.
<coro> = ac.wait(<tasks>)                # Accepts `return_when=ac.FIRST_COMPLETED`.
```

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

```
import asyncio as ac, collections as co, curses, curses.textpad, enum, random

P = co.namedtuple('P', 'x y')            # Position (x and y coordinates).
D = enum.Enum('D', 'n e s w')            # Direction (north, east, etc.).
W, H = 15, 7                             # Width and height of the field.

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

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

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

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

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

async def view(state, scr):
    x, y = curses.COLS//2 - W//2, curses.LINES//2 - H//2
    while True:
        scr.erase()
        curses.textpad.rectangle(scr, y-1, x-1, y+H, x+W)
        for id_, p in state.items():
            dx, dy = p.x - state['*'].x + W//2, p.y - state['*'].y + H//2
            scr.addstr(y + (dy % H), x + (dx % W), str(id_))
        scr.refresh()
        await ac.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=None)  # Accepts plt.plot(y_data).
plt.legend()                                      # Adds a legend of labels.
plt.title/xlabel/ylabel(<str>)                    # Adds title or axis label.
plt.show()                                        # Also plt.savefig(<path>).
plt.clf()                                         # Clears the plot (figure).
```

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_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT

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_UP) and (selected > 0)
        selected += (ch == KEY_DOWN) and (selected < len(paths)-1)
        first -= (first > selected)
        first += (first < selected-(height-1))
        if ch in [KEY_LEFT, KEY_RIGHT, ord('\n')]:
            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

Runs a desktop app for converting metric weights into pounds:

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

field = sg.Input(default_text='100', enable_events=True, key='QUANTITY')
menu = sg.Drop(['g', 'kg', 't'], 'kg', readonly=True, enable_events=True, k='UNIT')
text = sg.Text('is 220.462 lbs.', key='RESULT')
win = sg.Window('GUI App', [[field, menu], [text], [sg.Button('Close')]])

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

Scraping

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

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

get = lambda url: requests.get(url, headers={'User-Agent': 'cpc-bot'})
response = 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', string='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(get(f'https:{logo_url}').content)
print(f'URL: {python_url}, logo: file://{os.path.abspath(filename)}')
```

Selenium

Library for scraping websites with dynamic content.

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

```
<Drv> = webdriver.Chrome/Firefox/Safari()      # Opens the browser. Also `<Driver>.quit()`.
<Drv>.implicitly_wait(seconds)                 # Sets timeout for find_element/s() methods.
<Drv>.get('<url>')                             # Blocks until browser fires the load event.
<str> = <Drv>.page_source                      # Returns HTML of the page's current state.
<El>  = <Drv/El>.find_element('xpath', <str>)  # Accepts '//<tag>[@<attr_name>="<val>"]…'.
<str> = <El>.get_attribute('<name>')           # Returns attribute or a property if exists.
<El>.click/clear()                             # Also <El>.text and <El>.send_keys(<str>).
```

XPath — available in browser's console via `'$x("<xpath>")'`:

```
<xpath>   = //<element>[/ or // <element>]     # E.g. …/child, …//descendant, …/../sibling.
<xpath>   = //<el>/following-sibling::<el>     # Looks under first element. Also parent::.
<element> = <tag><conditions><index>           # Tag accepts */a/…. Use [1/2/…] for index.
<condit.> = [<sub_con> [and/or <sub_con>]]     # Use not(<sub_con>) to negate subcondition.
<sub_con> = @<attr>[="<val>"]                  # `text()=` and `.=` match (complete) text.
<sub_con> = contains(@<attr>, "<val>")         # Is <val> a substring of attribute's value?
<sub_con> = <element>                          # Has matching child? Descendant if //<el>.
```

Web App

Flask is a micro web framework that also includes a simple WSGI/HTTP 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 app object. Put at the top.
app.run(host=None, port=None, debug=None)   # Or `$ flask --app FILE run --ARG=…`.
```

- **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 to get better security.**
- **Debug mode restarts the app whenever script changes and displays errors in the browser.**

Serving Files

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

Serving HTML

```
@app.route('/<sport>')
def serve_html(sport):
    return fl.render_template_string('<h1>{{t}}</h1>', t=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 and `'fl.session.clear()'` clears it. A session cookie key needs to be set at the startup with `'app.secret_key = <str>'`.**

Serving JSON

```
@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'
>>> resp = requests.post(url, data={'team': 'Arsenal FC'})
>>> resp.json()
{'team': 'Arsenal FC', 'odds': [2.09, 3.74, 3.68]}
```

Profiling

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

Timing a Snippet

```
>>> from timeit import timeit
>>> timeit('list(range(10_000))', number=1000, globals=globals())
0.19373
```

Profiling by Line

```
$ pip3 install line_profiler
$ echo '@profile
def main():
    a = list(range(10_000))
    b = set(range(10_000))
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(10_000))
     4         1        534.1    534.1     67.8      b = set(range(10_000))
```

Visualizations

```
$ apt install graphviz && pip3 install gprof2dot snakeviz  # Or install graphviz.exe.
$ tail -n +2 test.py > test.tmp && mv test.tmp test.py     # Removes the first line.
$ python3 -m cProfile -o test.prof test.py                 # Runs a tracing profiler.
$ gprof2dot -f pstats test.prof | dot -T png -o test.png   # Generates a call graph.
$ xdg-open test.png                                        # Displays the call graph.
$ snakeviz test.prof                                       # Displays a flame graph.
```

Sampling Profilers

pip3 install	Profiles	How to run	Lines	Live
pyinstrument	CPU	pyinstrument test.py	×	×
py-spy	CPU	py-spy top -- python3 test.py	×	✓
scalene	CPU & RAM	scalene test.py	✓	×
memray	RAM	memray run --live test.py	✓	✓

NumPy

Array manipulation library. Can run hundred times faster than equivalent Python code.

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

```
<array> = np.array(<list/list_of_lists/…>)          # NumPy array. Accepts `dtype=np.int64`.
<array> = np.load(<path/file>)                      # Save array with np.save(<path>, <arr>).
```

```
<array> = np.zeros/ones/empty(shape)                # Pass a tuple of ints (dimension sizes).
<array> = np.arange(from, to_exc, ±step)            # Also np.linspace(start, stop, length).
<array> = np.random.randint(from, to_exc, shape)    # Also random.uniform(low, high, shape).
```

```
<view>  = <array>.reshape(shape)                    # Also `<array>.shape = (<int>, [...])`.
<array> = <array>.flatten()                         # Returns 1d copy. Also <array>.ravel().
```

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

```
<array> = np.concat(<arrays>, axis=0)               # Links arrays along first axis (rows).
<array> = np.vstack/column_stack(<arrays>)          # A 1d array is treated as a row/column.
<array> = np.tile/repeat(<arr>, <int/s> [, axis])   # Tiles whole array or repeats 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

```
<object>   = <2d>[row_index, col_index]            # Or <3d>[<int>, <int>, <int>].
<1d_view>  = <2d>[row_index]                       # Or <3d>[<int>, <int>, <slice>].
<1d_view>  = <2d>[:, col_index]                    # Or <3d>[<int>, <slice>, <int>].
<2d_view>  = <2d>[row_i:to_exc, col_i:to_exc]      # Or <3d>[<int>, <slice>, <slice>].
```

```
<1d_array> = <2d>[row_indices, col_indices]        # Or <3d>[<int/1d>, <1d>, <1d>].
<2d_array> = <2d>[row_indices]                     # Or <3d>[<int/1d>, <1d>, <slice>].
<2d_array> = <2d>[:, col_indices]                  # Or <3d>[<int/1d>, <slice>, <1d>].
<2d_array> = <2d>[np.ix_(row_is, col_is)]          # Or <3d>[<int/1d/2d>, <2d>, <2d>].
```

```
<2d_bools> = <2d> > <el/1d/2d>                     # A 1d object must be size of row.
<1/2d_arr> = <2d>[<2d/1d_bools>]                   # A 1d object must be size of col.
```

- **`':'` returns a slice of all dimension's indices. If dimension is omitted, it defaults to `':'`.**
- **Passing two slices (line 4) works the same as when a slice and 1d array are passed (line 7).**
- **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 two indices is passed.**
- **`'ix_([1, 2], [3, 4])'` returns `'[[1], [2]]'` and `'[[3, 4]]'`. Due to broadcasting rules, this is the same as indexing via `'[[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

Array reshaping procedure used by arithmetic operations, etc.

```
array_a = np.array([0.1,  0.6,  0.8])              # I.e. `array_a.shape == (3,)`.
array_b = np.array([[0.1], [0.6], [0.8]])          # I.e. `array_b.shape == (3, 1)`.
```

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

```
array_a = np.array([[0.1,  0.6,  0.8]])            # I.e. `array_a.shape == (1, 3)`.
array_b = np.array([[0.1], [0.6], [0.8]])          # I.e. `array_b.shape == (3, 1)`.
```

2. Expand dimensions with size 1 by duplicating their elements/arrays:

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

array_b = np.array([[0.1,  0.1,  0.1],             # I.e. `array_b.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('RGB', (width, heig))  # Creates an image. Also `color=<tuple_of_ints>`.
<Image> = Image.open(<path>)               # Identifies format based on the file's contents.
<Image> = <Image>.convert('<mode>')        # Converts the image to the new mode (see Modes).
<Image>.save(<path>)                       # Also `quality=<int>` if extension is jpg/jpeg.
<Image>.show()                             # Displays image in system's default preview app.
```

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

```
<Image> = <Image>.filter(<Filter>)         # Accepts ImageFilter.BLUR/SHARPEN/FIND_EDGES/….
<Image> = <Enhance>.enhance(<float>)       # E.g. `ImageEnhance.Contrast/Color/…(<Image>)`.
```

```
<array> = numpy.array(<Image>)             # Creates a 2d or 3d NumPy array from the image.
<Image> = Image.fromarray(<array>)         # Clip values with np.uint8(<arr>.clip(0, 255)).
```

Modes

- **`'L'` - Lightness (greyscale image). Each pixel is stored as an int between 0 and 255.**
- **`'RGB'` - Red, green, blue (true color image). Each pixel is a tuple of three integers.**
- **`'RGBA'` - RGB with alpha. Low alpha (i.e. fourth 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:

```
W, H = 100, 100
n_pixels = W * H
hues = (255 * i/n_pixels for i in range(n_pixels))
img = Image.new('HSV', (W, H))
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 i: max(0, min(255, i + 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>)           # An object for adding 2D graphics to the image.
<Draw>.point((x, y))                       # Draws a point. Accepts `fill=<int/tuple/str>`.
<Draw>.line((x1, y1, x2, y2 [, ...]))      # To get anti-aliasing use <Img>.resize((w, h)).
<Draw>.arc((x1, y1, x2, y2), deg1, deg2)   # Draws arc of an ellipse in clockwise direction.
<Draw>.rectangle((x1, y1, x2, y2))         # Also rounded_rectangle() and regular_polygon().
<Draw>.polygon((x1, y1, x2, y2, ...))      # The last point gets connected to the first one.
<Draw>.ellipse((x1, y1, x2, y2))           # To rotate it use <Image>.rotate(anticlock_deg).
<Draw>.text((x, y), <str>)                 # Accepts `font=ImageFont.truetype(path, size)`.
```

- **Pass `'fill=<color>'` to set primary color of the figure.**
- **Pass `'width=<int>'` to set the width of lines or contours.**
- **Pass `'outline=<color>'` to set the color of the contours.**
- **Color can be an int, tuple, `'#rrggbb[aa]'` or color name.**

Animation

Creates a GIF of a bouncing ball:

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

W, H, R = 126, 126, 10  # Width, Height, Radius.
frames = []
for velocity in range(1, 16):
    y = sum(range(velocity))
    frame = Image.new('L', (W, H))
    draw = ImageDraw.Draw(frame)
    draw.ellipse((W/2-R, y, W/2+R, y+2*R), 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 specified 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 how many bytes are in sample.
<tuple> = <Wave>.getparams()               # Returns namedtuple of all parameters.
<bytes> = <Wave>.readframes(<int>)         # Returns all frames if `-1` is passed.
```

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

- **The bytes object contains a sequence of frames, each consisting of one or more samples.**
- **In stereo signal, first sample of a frame belongs to the left channel (second to the right).**
- **Each sample consists of one or more bytes (depending on sample width) that, when converted to an integer, indicate the displacement of a speaker membrane at that moment.**
- **Integers should be encoded unsigned if sample width is one byte. For other sample sizes they should be encoded signed with little-endian byte order (least significant byte first).**

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)
    samples_b = (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 samples_b], p
```

Write Float Samples to WAV File

```
def write_to_wav_file(filename, samples_f, p=None, nchannels=1, sampwidth=2, fs=44100):
    def get_bytes(a_float):
        a_float = max(-1, min(1 - 2e-16, 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, fs, 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 sin, pi
get_sin = lambda i: sin(440 * pi*2 * i/44100) * 0.2
write_to_wav_file('test.wav', (get_sin(i) for i in range(100_000)))
```

Adds noise to the WAV file:

```
from random import uniform
samples_f, prms = read_wav_file('test.wav')
samples_f = (f + uniform(-0.02, 0.02) for f in samples_f)
write_to_wav_file('test.wav', samples_f, p=prms)
```

Audio Player

```
# $ pip3 install nava
from nava import play
play('test.wav')
```

Text to Speech

```
# $ pip3 install piper-tts sounddevice
import os, piper, sounddevice
os.system('python3 -m piper.download_voices en_US-lessac-high')
voice = piper.PiperVoice.load('en_US-lessac-high.onnx')
for sentence in voice.synthesize('Sally sells seashells by the seashore.'):
    sounddevice.wait()
    sounddevice.play(sentence.audio_float_array, sentence.sample_rate)
sounddevice.wait()
```

Synthesizer

Plays Popcorn by Gershon Kingsley:

```
# $ pip3 install numpy sounddevice
import itertools as it, math, numpy as np, sounddevice

def play_notes(notes, bpm=132, fs=44100, volume=0.1):
    beat_len  = 60/bpm * fs
    get_pause = lambda beats: it.repeat(0, int(beats * beat_len))
    get_sinus = lambda hz, i: math.sin(hz * math.pi*2 * i/fs) * volume
    get_wave  = lambda hz, beats: (get_sinus(hz, i) for i in range(int(beats * beat_len)))
    get_hertz = lambda note: 440 * 2 ** ((int(note[:2]) - 69) / 12)
    get_beats = lambda note: 1/2 if '♩' in note else 1/4 if '♪' in note else 1
    get_samps = lambda n: get_wave(get_hertz(n), get_beats(n)) if n else get_pause(1/4)
    samples_f = it.chain(get_pause(1/2), *(get_samps(n) for n in notes.split(',')))
    sounddevice.play(np.fromiter(samples_f, np.float32), fs, blocking=True)

play_notes('83♩,81♪,,83♪,,78♪,,74♪,,78♪,,71♪,,,,83♪,,81♪,,83♪,,78♪,,74♪,,78♪,,71♪,,,,'
           '83♩,85♪,,86♪,,85♪,,86♪,,83♪,,85♩,83♪,,85♪,,81♪,,83♪,,81♪,,83♪,,79♪,,83♪,,,,')
```

Pygame

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

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

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

Rect

Stores top-left corner, width and height.

```
<Rect> = pg.Rect(x, y, width, height)       # Creates Rect object. Truncates passed floats.
<int>  = <Rect>.x/y/centerx/centery          # Also `top`, `right`, etc. Allows assignments.
<tup.> = <Rect>.topleft/center               # Also `topright/bottomright/bottomleft/size`.
<Rect> = <Rect>.move((dx, dy))               # Use move_ip() to move the rectangle in-place.
```

```
<bool> = <Rect>.collidepoint((x, y))         # Returns True if rectangle contains the point.
<bool> = <Rect>.colliderect(<Rect>)          # Returns True if the rectangles are colliding.
<int>  = <Rect>.collidelist(<Rects>)         # Returns index of first colliding Rect or -1.
<list> = <Rect>.collidelistall(<Rects>)      # Returns indices of all colliding rectangles.
```

Surface

Stores image or main window's surface.

```
<Surf> = pg.Surface((w, h))                  # New RGB surface. RGBA if `flags=pg.SRCALPHA`.
<Surf> = pg.display.set_mode((w, h))         # Opens new window and returns surface object.
<Surf> = pg.image.load(<path/file>)          # Loads the image. Also get_width/get_height().
<Surf> = <Surf>.subsurface(<Rect>)           # Creates a new surface object from the cutout.
<view> = <Surf>.get_view()                   # Use <view>.write(<array>) to write to image.
```

```
<Surf>.fill(color)                           # Pass tuple of ints or pg.Color('<name/hex>').
<Surf>.set_at((x, y), color)                 # Updates a pixel. Also <Surf>.get_at((x, y)).
<Surf>.blit(<Surf>, (x, y))                  # Draws passed surface at a specified location.
```

```
<Surf> = tr.scale(<Surf>, (w, h))            # Import with `import pygame.transform as tr`.
<Surf> = tr.rotate(<Surf>, degrees)          # Rotates the surface for counterclock degrees.
<Surf> = tr.flip(<Surf>, flip_x=True)        # Mirrors over the y axis. Also `flip_y=True`.
```

```
line(<Surf>, color, (x1, y1), (x2, y2))      # Also aaline. Run `from pygame.draw import *`.
arc(<Surf>, color, <Rect>, rad1, rad2)       # Draws an arc of an ellipse counterclockwise.
rect(<Surf>, color, <Rect>, width=0)         # Also polygon(<Surf>, color, points, width=0).
circle(<Surf>, color, (x, y), radius)        # Also ellipse(<Surf>, color, <Rect>, width=0).
```

```
<Font> = pg.font.Font(<path/file>, size)     # Loads a TTF file. Pass None for default font.
<Surf> = <Font>.render(<str>, True, color)   # Accepts background color via fourth argument.
```

Sound

```
<Sound> = pg.mixer.Sound(<path/file>)        # Accepts WAV file or array of short integers.
<Sound>.play/stop()                          # Accepts `loops=-1`. Also set_volume(<float>).
```

Basic Mario Brothers Example

```
import pygame as pg, dataclasses as dc, enum, io, itertools, random as r, urllib.request

W, H, D = 50, 50, enum.Enum('D', 'n e s w')  # Width, Height, Direction.

def main():
    def get_window():
        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(20)]
    def get_mario():
        Mario = dc.make_dataclass('Mario', ['rect', 'vx', 'vy', 'dir', 'img_i'])
        return Mario(get_rect(1, 1), 0, 0, D.e, itertools.cycle(range(3)))
    def get_tiles():
        is_border = lambda x, y: x in [0, W-1] or y in [0, H-1]
        borders = [(x, y) for x in range(W) for y in range(H) if is_border(x, y)]
        platforms = [(r.randint(1, W-2), r.randint(2, H-2)) for _ in range(200)]
        return [get_rect(x, y) for x, y in borders + platforms]
    def get_rect(x, y):
        return pg.Rect(x*16, y*16, 16, 16)
    run(get_window(), get_images(), get_mario(), get_tiles())

def run(window, 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_velocity(mario, tiles, pressed)
        update_position(mario, tiles)
        draw(window, images, mario, tiles)
    pg.quit()

def update_velocity(mario, tiles, pressed):
    mario.vx += 2 * ((pg.K_RIGHT in pressed) - (pg.K_LEFT in pressed))
    mario.vx += (mario.vx < 0) - (mario.vx > 0)
    mario.vx = max(-4, min(4, mario.vx))
    mario.vy += 1 if is_airborne(mario, tiles) else (pg.K_UP in pressed) * -10

def update_position(mario, tiles):
    x, y = mario.rect.topleft
    steps = max(abs(mario.vx), abs(mario.vy))
    for _ in range(steps):
        bounds = get_boundaries(mario.rect, tiles)
        mario.vx, mario.vy = stop_on_collision(mario.vx, mario.vy, bounds)
        mario.rect.topleft = x, y = x + (mario.vx/steps), y + (mario.vy/steps)

def is_airborne(mario, tiles):
    return D.s not in get_boundaries(mario.rect, tiles)

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

def stop_on_collision(vx, vy, bounds):
    return (0 if (D.w in bounds and vx < 0) or (D.e in bounds and vx > 0) else vx,
            0 if (D.n in bounds and vy < 0) or (D.s in bounds and vy > 0) else vy)

def draw(window, images, mario, tiles):
    window.fill((85, 168, 255))
    mario.dir = mario.dir if mario.vx == 0 else D.w if mario.vx < 0 else D.e
    img_i = 4 if is_airborne(mario, tiles) else next(mario.img_i) if mario.vx else 6
    window.blit(images[img_i + ((mario.dir == D.w) * 9)], mario.rect)
    for tile in tiles:
        is_border = tile.x in [0, (W-1)*16] or tile.y in [0, (H-1)*16]
        window.blit(images[18 if is_border else 19], tile)
    pg.display.flip()

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

Pandas

Data analysis library. For examples see Plotly (p. 47).

```
# $ 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>)                        # Returns a series. Uses indices for 'index'.
<S>  = pd.Series(<dict>)                        # Returns a series. Uses 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] and 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 S of Period objects.
```

```
<S>.plot.line/area/bar/pie/hist()               # Generates a plot. Accepts `title=<str>` arg.
plt.show()                                      # Displays the plot. Also plt.savefig(<path>).
```

- **Use `'print(<S>.to_string())'` to print a Series that contains more than sixty items.**
- **Use `'<S>.index'` to get collection of keys and `'<S>.index = <coll>'` to update them.**
- **Only pass a list or Series to loc/iloc because `'obj[x, y]'` is converted to `'obj[(x, y)]'` and `'<S>.loc[key_1, key_2]'` is how you retrieve a value from a multi-indexed Series.**
- **Pandas uses NumPy types like `'np.int64'`. Series is converted to `'float64'` if np.nan is assigned to any item. Use `'<S>.astype(<str/type>)'` to get converted Series.**

Series — Aggregate, Transform, Map:

```
<el> = <S>.sum/max/mean/std/idxmax/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`
		`rank`
		`x  1.0`
		`y  2.0`
			`r  x  1.0`
			`y  2.0`

DataFrame

Table with labeled rows and columns.

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

```
<DF>   = pd.DataFrame(<list_of_rows>)          # Rows can be either lists, dicts or series.
<DF>   = pd.DataFrame(<dict_of_columns>)       # Columns can be either lists, dicts or series.
```

```
<el>   = <DF>.loc[row_key, col_key]            # Or: <DF>.iloc[row_i, col_i]
<S/DF> = <DF>.loc[row_key/s]                   # Or: <DF>.iloc[row_i/s]
<S/DF> = <DF>.loc[:, col_key/s]                # Or: <DF>.iloc[:, col_i/s]
<DF>   = <DF>.loc[row_bools, col_bools]        # Or: <DF>.iloc[row_bools, col_bools]
```

```
<S/DF> = <DF>[col_key/s]                       # Or: <DF>.<col_key>
<DF>   = <DF>[<S_of_bools>]                    # Filters rows. For example `df[df.x > 1]`.
<DF>   = <DF>[<DF_of_bools>]                   # Assigns NaN to items that are False in bools.
```

```
<DF>   = <DF> > <el/S/DF>                      # Returns DF of bools. Treats series as a row.
<DF>   = <DF> + <el/S/DF>                      # Items with non-matching keys get value NaN.
```

```
<DF>   = <DF>.set_index(col_key)               # Replaces row keys with column's values.
<DF>   = <DF>.reset_index(drop=False)          # Drops or moves row keys to column named index.
<DF>   = <DF>.sort_index(ascending=True)       # Sorts rows by row keys. Use `axis=1` for cols.
<DF>   = <DF>.sort_values(col_key/s)           # Sorts rows by passed column/s. Also `axis=1`.
```

```
<DF>   = <DF>.head/tail/sample(<int>)          # Returns first, last, or random n rows.
<DF>   = <DF>.describe()                       # Describes columns. Also info(), corr(), shape.
<DF>   = <DF>.query('<query>')                 # Filters rows. For example `df.query('x > 1')`.
```

```
<DF>.plot.line/area/bar/scatter(x=col_key, …)  # `y=col_key/s`. Also hist/box(column/by=col_k).
plt.show()                                     # Displays the plot. Also plt.savefig(<path>).
```

DataFrame — Merge, Join, Concat:

```
>>> 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 — Aggregate, Transform, Map:

```
<S>  = <DF>.sum/max/mean/std/idxmax/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`

- **Listed methods process the columns unless they receive `'axis=1'`. Exceptions to this rule are `'<DF>.dropna()'`, `'<DF>.drop(row_key/s)'` and `'<DF>.rename(<dict/func>)'`.**
- **Fifth result's columns are indexed with a multi-index. This means we need a tuple of column keys to specify a column: `'<DF>.loc[row_key, (col_key_1, col_key_2)]'`.**

Multi-Index

```
<DF> = <DF>.loc[row_key_1]                  # Also <DF>.loc[(slice(None), row_key_2), :].
<DF> = <DF>.loc[:, col_key_1]               # Same as <DF>.xs(col_key_1, axis=1, level=0).
<DF> = <DF>.set_index(col_key/s)            # Moves column/s to index. Also `append=True`.
<DF> = <DF>.pivot_table(index=col_key/s)    # `columns=key/s, values=k/s, aggfunc='mean'`.
<S>  = <DF>.stack/unstack(level=-1)         # Combines col. keys with index or vice versa.
```

File Formats

```
<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…>)  # Function read_hdf() accepts `key=<s/df_name>`.
<DF>   = pd.read_sql('<table/query>', <conn>)  # Pass SQLite3/Alchemy connection. See #SQLite.
```

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

- **`'$ pip3 install "pandas[excel]" odfpy lxml pyarrow'` installs dependencies.**
- **Csv functions use the same dialect as standard library's csv module (e.g. `'sep=","'`).**
- **Read_csv() only parses dates of columns that are listed in 'parse_dates'. It automatically tries to detect the format, but it can be helped with 'date_format' or 'dayfirst' arguments.**
- **We get a dataframe with DatetimeIndex if 'parse_dates' argument includes 'index_col'. Its `'resample("y/m/d/h")'` method returns Resampler object that is similar to GroupBy.**

GroupBy

Object that groups together rows of a dataframe based on the value of the passed column.

```
<GB> = <DF>.groupby(col_key/s)                 # Splits DF into groups based on passed col.
<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/std/idxmax/count()    # 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>)
```

Divides rows into groups and sums their columns. Result has a named index that creates column `'z'` on 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
```

Rolling

Object for rolling window calculations.

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

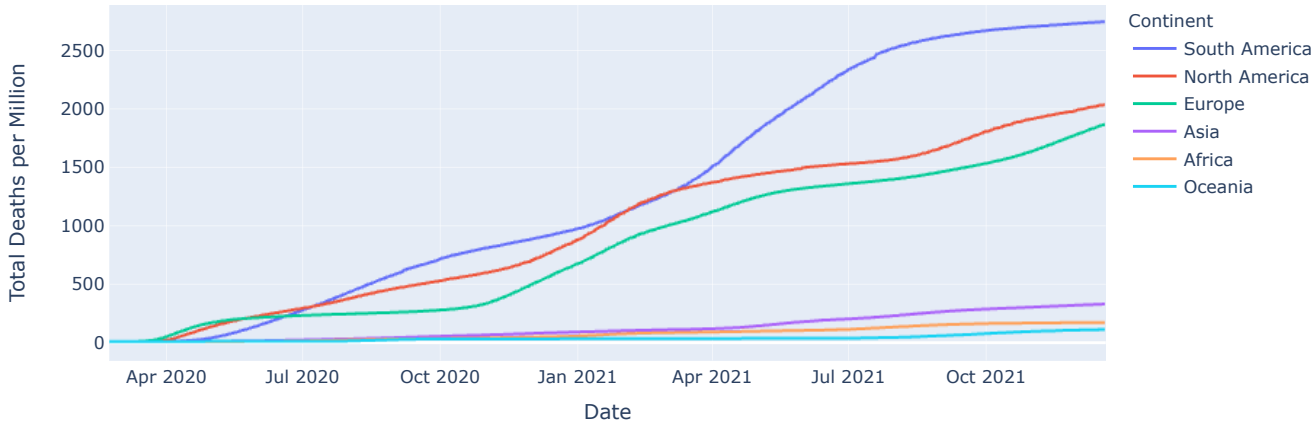
Plotly

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

```
<Fig> = px.line(<DF> [, y=col_key/s [, x=col_key]])   # Also px.line(y=<list> [, x=<list>]).
<Fig>.update_layout(paper_bgcolor='#rrggbb')          # Also `margin=dict(t=0, r=0, b=0, l=0)`.
<Fig>.write_html/json/image('<path>')                 # Use <Fig>.show() to display the plot.
```

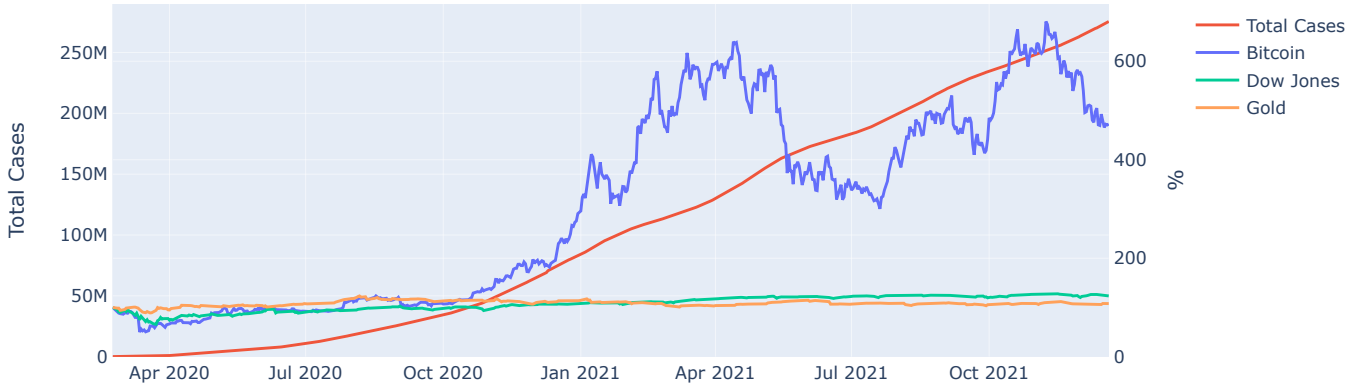
```
<Fig> = px.area/bar/box(<DF>, x=col_key, y=col_keys)  # Also `color=col_key`. All are optional.
<Fig> = px.scatter(<DF>, x=col_key, y=col_keys)       # Also `color/size/symbol=col_key`. Same.
<Fig> = px.scatter_3d(<DF>, x=col_key, y=col_key, …)  # `z=col_key`. Also color, size, symbol.
<Fig> = px.histogram(<DF>, x=col_keys, y=col_key)     # Also color, nbins. All are optional.
```

Displays a line chart of total COVID-19 deaths per million grouped by continent:




```
covid = pd.read_csv('https://raw.githubusercontent.com/owid/covid-19-data/8dde8ca49b'
                    '6e648c17dd420b2726ca0779402651/public/data/owid-covid-data.csv',
                    usecols=['iso_code', 'date', 'population', 'total_deaths'])
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')
```

Displays a multi-axis line chart of total COVID-19 cases and changes in prices of Bitcoin, Dow Jones and gold:




```
# $ pip3 install pandas lxml selenium plotly
import pandas as pd, selenium.webdriver, io, 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://catalog.ourworldindata.org/garden/covid/latest/compact/compact.csv'
    df = pd.read_csv(url, parse_dates=['date'])
    df = df[df.country == 'World']
    s = df.set_index('date').total_cases
    return s.rename('Total Cases')

def get_tickers():
    with selenium.webdriver.Chrome() as driver:
        driver.implicitly_wait(10)
        symbols = {'Bitcoin': 'BTC-USD', 'Gold': 'GC=F', 'Dow Jones': '%5EDJI'}
        return [get_ticker(driver, name, symbol) for name, symbol in symbols.items()]

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()
    html = io.StringIO(driver.page_source)
    dataframes = pd.read_html(html, 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 DF by joining columns on dates.
    df = df.sort_index().interpolate()            # Sorts rows by date, interpolates NaN-s.
    df = df.loc['2020-02-23':'2021-12-20']        # Keeps rows between the specified dates.
    df = (df / df.iloc[0]) * 100                  # Divides all cells by first day's value.
    df = df.join(covid)                           # Adds column that contains covid cases.
    return df.sort_values(df.index[-1], axis=1)   # Sorts columns by the 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=['#EF553B', '#636EFA', '#00CC96', '#FFA152'],
        legend=dict(x=1.08)
    )
    figure.show()

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

Appendix

Cython

Library that compiles Python-like code into C.

```
# $ pip3 install cython
import pyximport; pyximport.install()                # Module that runs Cython scripts.
import <cython_script>                               # Script must have '.pyx' extension.
```

All `'cdef'` definitions are optional, but they contribute to the speed-up:

```
cdef <type> <var_name> [= <obj/var>]                 # Either Python or C type variable.
cdef <ctype> *<pointer_name> [= &<var>]              # Use <pointer>[0] to get the value.
cdef <ctype>[size] <array_name> [= <coll/array>]     # Also `<ctype>[:] <mview> = <array>`.
cdef <ctype> *<array_name> [= <coll/array/pointer>]  # E.g. `<<ctype> *> malloc(n_bytes)`.
```

```
cdef <type> <func_name>(<type> [*]<arg_name>): ...   # Omitted types default to `object`.
```

```
cdef class <class_name>:                             # Also `cdef struct <struct_name>:`.
    cdef public <type> [*]<attr_name>                # Also `... <ctype> [*]<field_name>`.
    def __init__(self, <type> <arg_name>):           # Also `cdef __dealloc__(self):`.
        self.<attr_name> = <arg_name>                # Also `... free(<array/pointer>)`.
```

Virtual Environments

System for installing libraries directly into project's directory.

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

Basic Script Template

Run the script with `'$ python3 FILE'` or `'$ chmod u+x FILE; ./FILE'`. To automatically start the debugger when uncaught exception occurs run `'$ 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()
```

Index