

Comprehensive Python Cheatsheet

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



Contents

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

Main

```
if __name__ == '__main__':      # Runs main() if file wasn't imported.
    main()
```

List

```
<list> = <list>[<slice>]          # Or: <list>[from_inclusive : to_exclusive : ±step]

<list>.append(<el>)             # Or: <list> += [<el>]
<list>.extend(<collection>)     # Or: <list> += <collection>

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

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

- For details about sorted(), min() and max() see **sortable**.
- Module **operator** provides functions itemgetter() and mul() that offer the same functionality as lambda expressions above.

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

Dictionary

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

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

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

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

Counter

```
>>> from collections import Counter
>>> colors = ['blue', 'blue', 'blue', 'red', 'red']
>>> counter = Counter(colors)
>>> counter['yellow'] += 1
Counter({'blue': 3, 'red': 2, 'yellow': 1})
>>> counter.most_common()[0]
('blue', 3)
```

Set

```
<set> = set()                      # `{}` returns a dictionary.

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

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

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

Frozen Set

- Is immutable and hashable.
- That means it can be used as a key in a dictionary or as an element in a set.

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

Tuple

Tuple is an immutable and hashable list.

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

Named Tuple

Tuple's subclass with named elements.

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

Range

Immutable and hashable sequence of integers.

```
<range> = range(stop)                      # range(to_exclusive)  
<range> = range(start, stop)                # range(from_inclusive, to_exclusive)  
<range> = range(start, stop, ±step)         # range(from_inclusive, to_exclusive, ±step_size)  
  
>>> [i for i in range(3)]  
[0, 1, 2]
```

Enumerate

```
for i, el in enumerate(<collection> [, i_start]):  
    ...
```

Iterator

```
<iter> = iter(<collection>)  
<iter> = iter(<function>, to_exclusive)  
<el>   = next(<iter> [, default])  
<list> = list(<iter>)
```

```
# `iter(<iter>)` returns unmodified iterator.  
# A sequence of return values until 'to_exclusive'.  
# Raises StopIteration or returns 'default' on end.  
# Returns a list of iterator's remaining elements.
```

Itertools

```
import itertools as it
```

```

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

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

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

```

Generator

- Any function that contains a `yield` statement returns a generator.
- Generators and iterators are interchangeable.

```

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

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

```

Type

- Everything is an object.
- Every object has a type.
- Type and class are synonymous.

```

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

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. ABC can also manually decide whether or not a specific class is its virtual subclass, usually based on which methods the 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

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

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

<bool> = <sub_str> in <str>                         # Checks if string contains a substring.
<bool> = <str>.startswith(<sub_str>)              # Pass tuple of strings for multiple options.
<bool> = <str>.endswith(<sub_str>)                 # Pass tuple of strings for multiple options.
<int> = <str>.find(<sub_str>)                      # Returns start index of the first match or -1.
<int> = <str>.index(<sub_str>)                     # Same, but raises ValueError if missing.

<str> = <str>.replace(old, new [, count])           # Replaces 'old' with 'new' at most 'count' times.
<str> = <str>.translate(<table>)                  # Use `str.maketrans(<dict>)` to generate table.

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

- Also: '`lstrip()`', '`rstrip()`' and '`rsplit()`'.
- Also: '`lower()`', '`upper()`', '`capitalize()`' and '`title()`'.

Property Methods

	[!#\$%...]	[a-zA-Z]	[$\frac{1}{4}\frac{1}{2}\frac{3}{4}$]	[$^{2^3}1$]	[0-9]
isprintable()	✓	✓	✓	✓	✓
isalnum()		✓	✓	✓	✓
isnumeric()			✓	✓	✓
isdigit()				✓	✓
isdecimal()					✓

- Also: '`isspace()`' checks for '[\t\n\r\f\v\x1c-\x1f\x85\u2000...]'.

Regex

```
import re
<str> = re.sub(<regex>, new, text, count=0) # Substitutes all occurrences with 'new'.
<list> = re.findall(<regex>, text) # Returns all occurrences as strings.
<list> = re.split(<regex>, text, maxsplit=0) # Use brackets in regex to include the matches.
<Match> = re.search(<regex>, text) # Searches for first occurrence of the pattern.
<Match> = re.match(<regex>, text) # Searches only at the beginning of the text.
<iter> = re.finditer(<regex>, text) # Returns all occurrences as Match objects.
```

- Argument 'new' can be a function that accepts a Match object and returns a string.
- Search() and match() return None if they can't find a match.
- Argument '**flags=re.IGNORECASE**' can be used with all functions.
- Argument '**flags=re.MULTILINE**' makes '^' and '\$' match the start/end of each line.
- Argument '**flags=re.DOTALL**' makes dot also accept the '\n'.
- Use `r'\1'` or `\\\1` for backreference ('`\1`' returns a character with octal code 1).
- Add '?' after '*' and '+' to make them non-greedy.

Match Object

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

Special Sequences

```
'\d' == '[0-9]' # Matches decimal characters.
'\w' == '[a-zA-Z0-9_]' # Matches alphanumerics and underscore.
'\s' == '[ \t\n\r\f\v]' # Matches whitespaces.
```

- By default, decimal characters, alphanumerics and whitespaces from all alphabets are matched unless '**flags=re.ASCII**' argument is used.
- As shown above, it restricts all special sequence matches to the first 128 characters and prevents '`\s`' from accepting '`[\x1c-\x1f]`' (the so-called separator characters).
- Use a capital letter for negation (all non-ASCII characters will be matched when used in combination with ASCII flag).

Format

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

Attributes

```
>>> Person = collections.namedtuple('Person', 'name height')
>>> person = Person('Jean-Luc', 187)
>>> f'{person.height}'
'187'
>>> '{p.height}'.format(p=person)
'187'
```

General Options

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

- Options can be generated dynamically: `f'{<el>:{<str/int>}[...]}'`.
- Adding '`!r`' before the colon converts object to string by calling its `repr()` method.

Strings

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

Numbers

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

Floats

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

Comparison of presentation types:

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

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

- When both rounding up and rounding down are possible, the one that returns result with even last digit is chosen. That makes '`{6.5:.0f}`' a '6' and '`{7.5:.0f}`' an '8'.
- This rule only effects numbers that can be represented exactly by a float (.5, .25, ...).

Ints

```
{90:c} # 'Z'  
{90:b} # '1011010  
{90:X} # '5A'
```

Numbers

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

- `'int(<str>)' and 'float(<str>)' raise ValueError on malformed strings.`
 - Decimal numbers are stored exactly, unlike most floats where `'1.1 + 2.2 != 3.3'`.
 - Floats can be compared with: `'math.isclose(<float>, <float>)'`.
 - Precision of decimal operations is set with: `'decimal.getcontext().prec = <int>'`

Basic Functions

```

<num> = pow(<num>, <num>)                                # 0r: <num> ** <num>
<num> = abs(<num>)                                         # <float> = abs(<complex>)
<num> = round(<num> [, <ndigits>])                         # `round(126, -1) == 130`
```

Math

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

Statistics

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

Random

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

Bin, Hex

Bitwise Operators

Combinatorics

- Every function returns an iterator.
- If you want to print the iterator, you need to pass it to the list() function first!

```
import itertools as it

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

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

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

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

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

Datetime

- Module 'datetime' provides 'date' <D>, 'time' <T>, 'datetime' <DT> and 'timedelta' <TD> classes. All are immutable and hashable.
- Time and datetime objects can be 'aware' <a>, meaning they have defined timezone, or 'naive' <n>, meaning they don't.
- If object is naive, it is presumed to be in the system's timezone.

```
from datetime import date, time, datetime, timedelta
from dateutil.tz import UTC, tzlocal, gettz, datetime_exists, resolve_imaginary
```

Constructors

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

- Use '<D/DT>.weekday()' to get the day of the week as an int, with Monday being 0.
- '**fold=1**' means the second pass in case of time jumping back for one hour.
- Timedelta normalizes arguments to ±days, seconds (< 86 400) and microseconds (< 1M).

Now

```
<D/DTn> = D/DT.today()          # Current local date or naive datetime.
<DTn>    = DT.utcnow()           # Naive datetime from current UTC time.
<DTa>    = DT.now(<tzinfo>)     # Aware datetime from current tz time.
```

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

Timezone

```
<tzinfo> = UTC                                # UTC timezone. London without DST.  
<tzinfo> = tzlocal()                          # Local timezone. Also gettz().  
<tzinfo> = gettz('<Continent>/<City>')      # 'Continent/City_Name' timezone or None.  
<DTa>   = <DT>.astimezone(<tzinfo>)        # Datetime, converted to the passed timezone.  
<Ta/DTa> = <T/DT>.replace(tzinfo=<tzinfo>)  # Unconverted object with a new timezone.
```

Encode

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

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

Decode

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

Format

```
>>> dt = datetime.strptime('2015-05-14 23:39:00.00 +2000', '%Y-%m-%d %H:%M:%S.%f %z')  
>>> dt.strftime("%A, %dth of %B '%y, %I:%M%p %Z")  
"Thursday, 14th of May '15, 11:39PM UTC+02:00"
```

- '%Z' only accepts 'UTC/GMT' and local timezone's code. '%z' also accepts '±HH:MM'.
- For abbreviated weekday and month use '%a' and '%b'.

Arithmetics

```
<D/DT>   = <D/DT> ± <TD>                   # Returned datetime can fall into missing hour.  
<TD>     = <D/DTn> - <D/DTn>                # Returns the difference, ignoring time jumps.  
<TD>     = <DTa> - <DTa>                    # Ignores time jumps if they share tzinfo object.  
<TD>     = <TD> * <real>                   # Also: <TD> = abs(<TD>) and <TD> = <TD> ± <TD>.  
<float>  = <TD> / <TD>                     # How many weeks/years there are in TD. Also //.
```

Arguments

Inside Function Call

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

Inside Function Definition

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

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

Splat Operator

Inside Function Call

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

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

Is the same as:

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

Inside Function Definition

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

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

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

Legal argument combinations:

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

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

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

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

Other Uses

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

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

Inline

Lambda

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

Comprehensions

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

Map, Filter, Reduce

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

- Reduce must be imported from the `functools` module.

Any, All

```
<bool> = any(<collection>) # Is `bool(el)` True for any element.  
<bool> = all(<collection>) # Is True for all elements or empty.
```

Conditional Expression

```
<obj> = <exp> if <condition> else <exp> # Only one expression gets evaluated.  
  
>>> [a if a else 'zero' for a in (0, 1, 2, 3)]  
['zero', 1, 2, 3]
```

Named Tuple, Enum, Dataclass

```
from collections import namedtuple  
Point = namedtuple('Point', 'x y')  
point = Point(0, 0) # Creates a tuple's subclass.  
  
from enum import Enum  
Direction = Enum('Direction', 'n e s w')  
direction = Direction.n # Returns its instance.  
  
from dataclasses import make_dataclass  
Player = make_dataclass('Player', ['loc', 'dir']) # Creates a class.  
player = Player(point, direction) # Returns its instance.
```

Imports

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

- Package is a collection of modules, but it can also define its own objects.
- On a filesystem this corresponds to a directory of Python files with an optional init script.
- Running '`import <package>`' does not automatically provide access to the package's modules unless they are explicitly imported in its init script.

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 the nested function.

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

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

- If multiple nested functions within enclosing function reference the same value, that value gets shared.
- To dynamically access function's first free variable use '`<function>.__closure__[0].cell_contents`'.

Partial

```
from functools import partial
<function> = partial(<function> [, <arg_1>, <arg_2>, ...])

>>> import operator as op
>>> multiply_by_3 = partial(op.mul, 3)
>>> multiply_by_3(10)
30
```

- Partial is also useful in cases when function needs to be passed as an argument because it enables us to set its arguments beforehand.
- A few examples being: '`defaultdict(<function>)`', '`iter(<function>, to_exclusive)`' and dataclass's '`field(default_factory=<function>)`'.

Non-Local

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

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

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

Decorator

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

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

Debugger Example

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

```
from functools import wraps

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

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

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

LRU Cache

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

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    return n if n < 2 else fib(n-2) + fib(n-1)
```

- Default size of the cache is 128 values. Passing '**maxsize=None**' makes it unbounded.
- CPython interpreter limits recursion depth to 1000 by default. To increase it use '**sys.setrecursionlimit(<depth>)**'.

Parametrized Decorator

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

```
from functools import wraps

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

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

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

Class

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

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

- Return value of `repr()` should be unambiguous and of `str()` readable.
- If only `repr()` is defined, it will also be used for `str()`.
- Methods decorated with '`@staticmethod`' do not receive '`self`' nor '`cls`' as their first arg.

Str() use cases:

```
print(<el>)
f'<el>'
```

logging.warning(<el>)

csv.writer(<file>).writerow([<el>])

`raise` Exception(<el>)

Repr() use cases:

```
print/str/repr([<el>])
f'<el>!r'
```

Z = dataclasses.make_dataclass('Z', ['a']); print/str/repr(Z(<el>))

>>> <el>

Constructor Overloading

```
class <name>:
    def __init__(self, a=None):
        self.a = a
```

Inheritance

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

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

Multiple Inheritance

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

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

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

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'
```

Dataclass

Decorator that automatically generates init(), repr() and eq() special methods.

```
from dataclasses import dataclass, field

@dataclass(order=False, frozen=False)
class <class_name>:
    <attr_name_1>: <type>
    <attr_name_2>: <type> = <default_value>
    <attr_name_3>: list/dict/set = field(default_factory=list/dict/set)
```

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

Inline:

```
from dataclasses import make_dataclass
<class> = make_dataclass('<class_name>', <coll_of_attribute_names>)
<class> = make_dataclass('<class_name>', <coll_of_tuples>)
<tuple> = ('<attr_name>', <type> [, <default_value>])
```

Rest of type annotations (CPython interpreter ignores them all):

```
def func(<arg_name>: <type> [= <obj>]) -> <type>: ...
<var_name>: typing.List/Set/Iterable/Sequence/Optional[<type>]
<var_name>: typing.Dict/Tuple/Union[<type>, ...]
```

Slots

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

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

Copy

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

Duck Types

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

Comparable

- If eq() method is not overridden, it returns '`id(self) == id(other)`', which is the same as '`self is other`'.
- That means all objects compare not equal by default.
- Only the left side object has eq() method called, unless it returns NotImplemented, in which case the right object is consulted. False is returned if both return NotImplemented.
- Ne() automatically works on any object that has eq() defined.

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

Hashable

- Hashable object needs both hash() and eq() methods and its hash value should never change.
- Hashable objects that compare equal must have the same hash value, meaning default hash() that returns '`id(self)`' will not do.
- That is why Python automatically makes classes unhashable if you only implement eq().

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

Sortable

- With 'total_ordering' decorator, you only need to provide eq() and one of lt(), gt(), le() or ge() special methods and the rest will be automatically generated.
- Functions sorted() and min() only require lt() method, while max() only requires gt(). However, it is best to define them all so that confusion doesn't arise in other contexts.
- When two lists, strings or dataclasses are compared, their values get compared in order until a pair of unequal values is found. The comparison of this two values is then returned. The shorter sequence is considered smaller in case of all values being equal.

```
from functools import total_ordering

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

Iterator

- Any object that has methods next() and iter() is an iterator.
- Next() should return next item or raise StopIteration.
- Iter() should return 'self'.

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

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

Python has many different iterator objects:

- Sequence iterators returned by the iter() function, such as list_iterator and set_iterator.
- Objects returned by the itertools module, such as count, repeat and cycle.
- Generators returned by the generator functions and generator expressions.
- File objects returned by the open() function, etc.

Callable

- All functions and classes have a call() method, hence are callable.
- When this cheatsheet uses '<function>' as an argument, it actually means '<callable>'.

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

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

Context Manager

- Enter() should lock the resources and optionally return an object.
- Exit() should release the resources.
- Any exception that happens inside the with block is passed to the exit() method.
- If it wishes to suppress the exception it must return a true value.

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

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

Iterable Duck Types

Iterable

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

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

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

Collection

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

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

Sequence

- Only required methods are `len()` and `getitem()`.
- `Getitem()` should return an item at the passed index or raise `IndexError`.
- `Iter()` and `contains()` automatically work on any object that has `getitem()` defined.
- `Reversed()` automatically works on any object that has `len()` and `getitem()` defined.

```
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:

- Glossary defines iterable as any object with `iter()` or `getitem()` and sequence as any object with `getitem()` and `len()`. It does not define collection.
- Passing ABC Iterable to `isinstance()` or `issubclass()` checks whether object/class has method `iter()`, while ABC Collection checks for `iter()`, `contains()` and `len()`.

ABC Sequence

- It's a richer interface than the basic sequence.
- Extending it generates `iter()`, `contains()`, `reversed()`, `index()` and `count()`.
- Unlike '`abc.Iterable`' and '`abc.Collection`', it is not a duck type. That is why '`issubclass(MySequence, abc.Sequence)`' would return `False` even if `MySequence` had all the methods defined. It however recognizes `list`, `tuple`, `range`, `str`, `bytes`, `bytearray`, `memoryview` and `deque`, because they are registered as Sequence's virtual subclasses.

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

Table of required and automatically available special methods:

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

- Other ABCs that generate missing methods are: MutableSequence, Set, MutableSet, Mapping and MutableMapping.
- Names of their required methods are stored in '`<abc>.__abstractmethods__`'.

Enum

```
from enum import Enum, auto

class <enum_name>(Enum):
    <member_name_1> = <value_1>
    <member_name_2> = <value_2_a>, <value_2_b>
    <member_name_3> = auto()

    • If there are no numeric values before auto(), it returns 1.
    • Otherwise it returns an increment of the last numeric value.

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

list_of_members = list(<enum>)
member_names   = [a.name for a in <enum>]
member_values  = [a.value for a in <enum>]
random_member  = random.choice(list(<enum>))

def get_next_member(member):
    members = list(member.__class__)
    index  = (members.index(member) + 1) % len(members)
    return members[index]
```

Inline

```
Cutlery = Enum('Cutlery', 'fork knife spoon')
Cutlery = Enum('Cutlery', ['fork', 'knife', 'spoon'])
Cutlery = Enum('Cutlery', {'fork': 1, 'knife': 2, 'spoon': 3})
```

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

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

- Member names are in all caps because trying to access a member that is named after a reserved keyword raises SyntaxError.

Exceptions

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

Complex Example

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

- Code inside the '**else**' block will only be executed if '**try**' block had no exceptions.
- Code inside the '**finally**' block will always be executed (unless a signal is received).

Catching Exceptions

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

- Also catches subclasses of the exception.
- Use '**traceback.print_exc()**' to print the error message to stderr.
- Use '**print(<name>)**' to print just the cause of the exception (its arguments).
- Use '**logging.exception(<message>)**' to log the exception.

Raising Exceptions

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

Re-raising caught exception:

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

Exception Object

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

Built-in Exceptions

```
BaseException
└── SystemExit           # Raised by the sys.exit() function.
└── KeyboardInterrupt    # Raised when the user hits the interrupt key (ctrl-c).
└── Exception
    ├── ArithmeticError   # User-defined exceptions should be derived from this class.
    │   └── ZeroDivisionError # Base class for arithmetic errors.
    ├── AssertionError     # Raised when dividing by zero.
    ├── AttributeError      # Raised by `assert <exp>` if expression returns false value.
    ├── EOFError            # Raised when an attribute is missing.
    ├── LookupError          # Raised by input() when it hits end-of-file condition.
    │   └── IndexError        # Raised when a look-up on a collection fails.
    │   └── KeyError          # Raised when a sequence index is out of range.
    ├── MemoryError          # Raised when a dictionary key or set element is missing.
    ├── NameError             # Out of memory. Could be too late to start deleting vars.
    ├── OSError               # Raised when an object is missing.
    │   └── FileNotFoundError # Errors such as "file not found" or "disk full" (see Open).
    ├── RuntimeError          # When a file or directory is requested but doesn't exist.
    │   └── RecursionError    # Raised by errors that don't fall into other categories.
    ├── StopIteration          # Raised when the maximum recursion depth is exceeded.
    ├── TypeError              # Raised by next() when run on an empty iterator.
    ├── ValueError             # Raised when an argument is of wrong type.
    └── UnicodeError          # When an argument is of right type but inappropriate value.
                                # Raised when encoding/decoding strings to/from bytes fails.
```

Collections and their exceptions:

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

Useful built-in exceptions:

```
raise TypeError('Argument is of wrong type!')
raise ValueError('Argument is of right type but inappropriate value!')
raise RuntimeError('None of above!')
```

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(<el>)       # Prints to stderr and exits with 1.
sys.exit(<int>)       # Exits with passed exit code.
```

Print

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

- Use '**file=sys.stderr**' for messages about errors.

- Use '**flush=True**' to forcibly flush the stream.

Pretty Print

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

- Levels deeper than 'depth' get replaced by '...'.

Input

Reads a line from user input or pipe if present.

```
<str> = input(prompt=None)
```

- Trailing newline gets stripped.
- Prompt string is printed to the standard output before reading input.
- Raises **EOFError** when user hits EOF (ctrl-d/ctrl-z) or input stream gets exhausted.

Command Line Arguments

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

Argument Parser

```
from argparse import ArgumentParser, FileType
p = ArgumentParser(description=<str>)
p.add_argument('<short_name>', '--<name>', action='store_true') # Flag.
p.add_argument('<short_name>', '--<name>', type=<type>)          # Option.
p.add_argument('<name>', type=<type>, nargs=1)                   # First argument.
p.add_argument('<name>', type=<type>, nargs='+')                 # Remaining arguments.
p.add_argument('<name>', type=<type>, nargs='*')                  # Optional arguments.
args = p.parse_args()                                         # Exits on error.
value = args.<name>
```

- Use '**help=<str>**' to set argument description that will be displayed in help message.
- Use '**default=<el>**' to set the default value.
- Use '**type=FileType(<mode>)**' for files. Accepts 'encoding', but 'newline' is None.

Open

Opens the file and returns a corresponding file object.

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

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

Modes

- '**r**' - Read (default).
- '**w**' - Write (truncate).
- '**x**' - Write or fail if the file already exists.
- '**a**' - Append.
- '**w+**' - Read and write (truncate).
- '**r+**' - Read and write from the start.
- '**a+**' - Read and write from the end.
- '**t**' - Text mode (default).
- '**b**' - Binary mode ('**br**', '**bw**', '**bx**', ...).

Exceptions

- '**FileNotFoundException**' can be raised when reading with '**r**' or '**r+**'.
- '**FileExistsError**' can be raised when writing with '**x**'.
- '**IsADirectoryError**' and '**PermissionError**' can be raised by any.
- '**OSError**' is the parent class of all listed exceptions.

File Object

```
<file>.seek(0)                      # Moves to the start of the file.  
<file>.seek(offset)                 # Moves 'offset' chars/bytes from the start.  
<file>.seek(0, 2)                   # Moves to the end of the file.  
<bin_file>.seek(±offset, <anchor>)  # Anchor: 0 start, 1 current position, 2 end.  
  
<str/bytes> = <file>.read(size=-1)  # Reads 'size' chars/bytes or until EOF.  
<str/bytes> = <file>.readline()      # Returns a line or empty string/bytes on EOF.  
<list>     = <file>.readlines()       # Returns a list of remaining lines.  
<str/bytes> = next(<file>)         # Returns a line using buffer. Do not mix.  
  
<file>.write(<str/bytes>)          # Writes a string or bytes object.  
<file>.writelines(<collection>)    # Writes a coll. of strings or bytes objects.  
<file>.flush()                    # Flushes write buffer. Runs every 4096/8192 B.
```

- Methods do not add or strip trailing newlines, 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

```
from os import.getcwd, path, listdir, scandir  
from glob import glob  
  
<str>  = getcwd()                  # Returns the current working directory.  
<str>  = path.join(<path>, ...)   # Joins two or more pathname components.  
<str>  = path.abspath(<path>)     # Returns absolute path.  
  
<str>  = path.basename(<path>)    # Returns final component of the path.  
<str>  = path.dirname(<path>)     # Returns path without the final component.  
<tup.> = path.splitext(<path>)  # Splits on last period of the final component.
```

```

<list> = listdir(path='.')
<list> = glob('<pattern>')

<bool> = path.exists(<path>)           # Returns filenames located at path.
<bool> = path.isfile(<path>)          # Returns paths matching the wildcard pattern.

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

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

```

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

```

Path Object

```

from pathlib import Path

<Path> = Path(<path> [, ...])      # Accepts strings, Paths and DirEntry objects.
<Path> = <path> / <path> [/ ...]    # First or second path must be a Path object.

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

<Path> = <Path>.parent           # Returns Path without the final component.
<str> = <Path>.name              # Returns final component as a string.
<str> = <Path>.stem               # Returns final component without extension.
<str> = <Path>.suffix             # Returns final component's extension.
<tup.> = <Path>.parts            # Returns all components as strings.

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

<str> = str(<Path>)             # Returns path as a string.
<file> = open(<Path>)           # Also <Path>.read/write_text/bytes().

```

OS Commands

```

import os, shutil, subprocess

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

shutil.copy(from, to)              # Copies the file. 'to' can exist or be a dir.
shutil.copytree(from, to)          # Copies the directory. 'to' must not exist.

os.rename(from, to)                # Renames/moves the file or directory.
os.replace(from, to)              # Same, but overwrites 'to' if it exists.

```

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

- Paths can be either strings, Paths or DirEntry objects.
- Functions report OS related errors by raising either OSError or one of its subclasses.

Shell Commands

```
<pipe> = os.popen('<command>')    # Executes command in sh/cmd. Returns its stdout pipe.  
<str>  = <pipe>.read(size=-1)      # Reads 'size' chars or until EOF. Also readline/s().  
<int>  = <pipe>.close()          # Closes the pipe. Returns None on success.
```

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

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

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

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

JSON

Text file format for storing collections of strings and numbers.

```
import json  
<str>   = json.dumps(<object>)      # Converts object to JSON string.  
<object> = json.loads(<str>)        # Converts JSON string to object.
```

Read Object from JSON File

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

Write Object to JSON File

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

Pickle

Binary file format for storing Python objects.

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

Read Object from File

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

Write Object to File

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

CSV

Text file format for storing spreadsheets.

```
import csv
```

Read

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

- File must be opened with a '`'newline='''`' argument, or newlines embedded inside quoted fields will not be interpreted correctly!
- To print the spreadsheet to the console use `Tabulate` library.
- For XML and binary Excel files (xlsx, xlsm and xlsb) use `Pandas` library.

Write

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

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

Parameters

- **'dialect'** - Master parameter that sets the default values. String or a Dialect object.
- **'delimiter'** - A one-character string used to separate fields.
- **'quotechar'** - Character for quoting fields that contain special characters.
- **'doublequote'** - Whether quotechars inside fields are/get doubled or escaped.
- **'skipinitialspace'** - Is space character at the start of the field stripped by the reader.
- **'lineterminator'** - How writer terminates rows. Reader is hardcoded to '\n', '\r', '\r\n'.
- **'quoting'** - 0: As necessary, 1: All, 2: All but numbers which are read as floats, 3: None.
- **'escapechar'** - Character for escaping quotechars if doublequote is False.

Dialects

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

Read Rows from CSV File

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

Write Rows to CSV File

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

SQLite

A server-less database engine that stores each database into a separate file.

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

Read

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

Write

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

Or:

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

Placeholders

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

- Passed values can be of type str, int, float, bytes, None, bool, datetime.date or datetime.datetime.
- Bools will be stored and returned as ints and dates as ISO formatted strings.

Example

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

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

MySQL

Has a very similar interface, with differences listed below.

```
# $ pip3 install mysql-connector  
from mysql import connector  
<conn> = connector.connect(host=<str>, ...)      # `user=<str>, password=<str>, database=<str>`.  
<cursor> = <conn>.cursor()                      # Only cursor has execute() method.  
<cursor>.execute('<query>')                     # Can raise a subclass of connector.Error.  
<cursor>.execute('<query>', <list/tuple>)       # Replaces '%s's in query with values.  
<cursor>.execute('<query>', <dict/namedtuple>)  # Replaces '%(<key>)s's with values.
```

Bytes

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

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

Encode

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

Decode

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

Read Bytes from File

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

Write Bytes to File

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

Struct

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

```
from struct import pack, unpack
<bytes> = pack('<format>', <el_1> [, ...]) # Packages arguments into bytes object.
<tuple> = unpack('<format>', <bytes>) # Use iter_unpack() for iterator of tuples.
```

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

Format

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

- '=' - System's byte order (usually little-endian).
- '<' - Little-endian.
- '>' - 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.

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

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

Floating point types:

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

Array

List that can only hold numbers of a predefined type. Available types and their minimum sizes in bytes are listed above. Sizes and byte order are always determined by the system.

```
from array import array
<array> = array('<typecode>', <collection>)      # Array from collection of numbers.
<array> = array('<typecode>', <bytes>)           # Array from bytes object.
<array> = array('<typecode>', <array>)            # Treats array as a sequence of numbers.
<bytes> = bytes(<array>)                          # Or: <array>.tobytes()
<file>.write(<array>)                           # Writes array to the binary file.
```

Memory View

- A sequence object that points to the memory of another object.
- Each element can reference a single or multiple consecutive bytes, depending on format.
- Order and number of elements can be changed with slicing.
- Casting only works between char and other types and uses system's sizes.
- Byte order is always determined by the system.

```
<mview> = memoryview(<bytes/bytearray/array>)    # Immutable if bytes, else mutable.
<real> = <mview>[<index>]                      # Returns an int or a float.
<mview> = <mview>[<slice>]                      # Mview with rearranged elements.
<mview> = <mview>.cast('<typecode>')            # Casts memoryview to the new format.
<mview>.release()                                # Releases the object's memory buffer.
```

Decode

```
<bytes> = bytes(<mview>)                         # Returns a new bytes object.
<bytes> = <bytes>.join(<coll_of_mvviews>)        # Joins mvviews using bytes object as sep.
<array> = array('<typecode>', <mview>)            # Treats mview as a sequence of numbers.
<file>.write(<mview>)                           # Writes mview to the binary file.

<list> = list(<mview>)                           # Returns a list of ints or floats.
<str> = str(<mview>, 'utf-8')                     # Treats mview as a bytes object.
<int> = int.from_bytes(<mview>, ...)            # `byteorder='little/big', signed=False` .
'<hex>' = <mview>.hex()                          # Treats mview as a bytes object.
```

Deque

A thread-safe list with efficient appends and pops from either side. Pronounced "deck".

```
from collections import deque
<deque> = deque(<collection>, maxlen=None)

<deque>.appendleft(<el>)                         # Opposite element is dropped if full.
<deque>.extendleft(<collection>)                  # Collection gets reversed.
<el> = <deque>.popleft()                        # Raises IndexError if empty.
<deque>.rotate(n=1)                                # Rotates elements to the right.
```

Threading

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

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

Thread

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

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

Lock

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

Or:

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

Semaphore, Event, Barrier

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

Thread Pool Executor

- Object that manages thread execution.
- An object with the same interface called ProcessPoolExecutor provides true parallelism by running a separate interpreter in each process. All arguments must be **pickable**.

```
<Exec> = ThreadPoolExecutor(max_workers=None)    # Or: `with ThreadPoolExecutor() as <name>: ...`
<Exec>.shutdown(wait=True)                         # Blocks until all threads finish executing.
```

```
<iter> = <Exec>.map(<func>, <args_1>, ...)
<Futr> = <Exec>.submit(<func>, <arg_1>, ...)
<bool> = <Futr>.done()                           # A multithreaded and non-lazy map().
<obj>  = <Futr>.result()                          # Starts a thread and returns its Future object.
                                                       # Checks if the thread has finished executing.
                                                       # Waits for thread to finish and returns result.
```

Queue

A thread-safe FIFO queue. For LIFO queue use LifoQueue.

```
from queue import Queue
<Queue> = Queue(maxsize=0)
```

```
<Queue>.put(<el>)                                # Blocks until queue stops being full.
<Queue>.put_nowait(<el>)                           # Raises queue.Full exception if full.
<el> = <Queue>.get()                               # Blocks until queue stops being empty.
<el> = <Queue>.get_nowait()                         # Raises queue.Empty exception if empty.
```

Operator

Module of functions that provide the functionality of operators.

```
import operator as op
<el>      = op.add/sub/mul/truediv/floordiv/mod(<el>, <el>) # +, -, *, /, //, %
<int/set> = op.and_/or_/xor(<int/set>, <int/set>)           # &, |, ^
<bool>    = op.eq/ne/lt/le/gt/ge(<sortable>, <sortable>)       # ==, !=, <, <=, >, >=
<func>    = op.itemgetter/attrgetter/methodcaller(<obj>)        # [index/key], .name, .name()
```



```
elementwise_sum = map(op.add, list_a, list_b)
sorted_by_second = sorted(<collection>, key=op.itemgetter(1))
sorted_by_both = sorted(<collection>, key=op.itemgetter(1, 0))
product_of_elems = functools.reduce(op.mul, <collection>)
union_of_sets = functools.reduce(op.or_, <coll_of_sets>)
first_element = op.methodcaller('pop', 0)(<list>)
```

- Binary operators require objects to have `and()`, `or()`, `xor()` and `invert()` special methods, unlike logical operators that work on all types of objects.
- Also: '`<bool> = <bool> &|^ <bool>`' and '`<int> = <bool> &|^ <int>`'.

Introspection

Inspecting code at runtime.

Variables

```
<list> = dir()                                # Names of local variables (incl. functions).
<dict> = vars()                               # Dict of local variables. Also locals().
<dict> = globals()                            # Dict of global variables.
```

Attributes

```
<list> = dir(<object>)                      # Names of object's attributes (incl. methods).
<dict> = vars(<object>)                     # Dict of writable attributes. Also <obj>.__dict__.
<bool> = hasattr(<object>, '<attr_name>')   # Checks if getattr() raises an AttributeError.
value = getattr(<object>, '<attr_name>')     # Raises AttributeError if attribute is missing.
setattr(<object>, '<attr_name>', value)       # Only works on objects with '__dict__' attribute.
delattr(<object>, '<attr_name>')               # Same. Also del <object>.attr_name.
```

Parameters

```
<Sig>  = inspect.signature(<function>)          # Function's Signature object.
<dict> = <Sig>.parameters                     # Dict of Parameter objects.
<memb> = <Param>.kind                         # Member of ParameterKind enum.
<obj>  = <Param>.default                      # Default value or <Param>.empty.
<type> = <Param>.annotation                  # Type or <Param>.empty.
```

Metaprogramming

Code that generates code.

Type

Type is the root class. If only passed an object it returns its type (class). Otherwise it creates a new class.

```
<class> = type('<class_name>', <tuple_of_parents>, <dict_of_class_attributes>)
```

```
>>> Z = type('Z', (), {'a': 'abcde', 'b': 12345})  
>>> z = Z()
```

Meta Class

A class that creates classes.

```
def my_meta_class(name, parents, attrs):  
    attrs['a'] = 'abcde'  
    return type(name, parents, attrs)
```

Or:

```
class MyMetaClass(type):  
    def __new__(cls, name, parents, attrs):  
        attrs['a'] = 'abcde'  
        return type.__new__(cls, name, parents, attrs)
```

- `New()` is a class method that gets called before `init()`. If it returns an instance of its class, then that instance gets passed to `init()` as a 'self' argument.
- It receives the same arguments as `init()`, except for the first one that specifies the desired type of the returned instance (`MyMetaClass` in our case).
- Like in our case, `new()` can also be called directly, usually from a `new()` method of a child class (`def __new__(cls): return super().__new__(cls)`).
- The only difference between the examples above is that `my_meta_class()` returns a class of type `type`, while `MyMetaClass()` returns a class of type `MyMetaClass`.

Metaclass Attribute

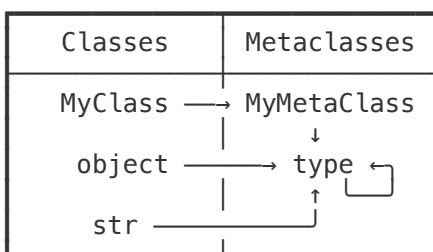
Right before a class is created it checks if it has the 'metaclass' attribute defined. If not, it recursively checks if any of his parents has it defined and eventually comes to `type()`.

```
class MyClass(metaclass=MyMetaClass):  
    b = 12345
```

```
>>> MyClass.a, MyClass.b  
('abcde', 12345)
```

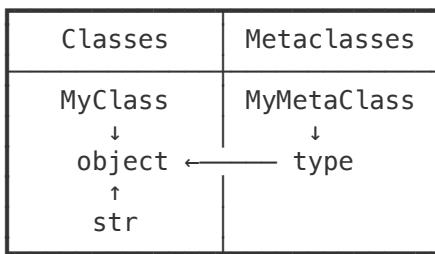
Type Diagram

```
type(MyClass) == MyMetaClass          # MyClass is an instance of MyMetaClass.  
type(MyMetaClass) == type            # MyMetaClass is an instance of type.
```



Inheritance Diagram

```
MyClass.__base__ == object          # MyClass is a subclass of object.  
MyMetaClass.__base__ == type        # MyMetaClass is a subclass of type.
```



Eval

```
>>> from ast import literal_eval
>>> literal_eval('[1, 2, 3]')
[1, 2, 3]
>>> literal_eval('1 + 2')
ValueError: malformed node or string
```

Coroutines

- Coroutines have a lot in common with threads, but unlike threads, they only give up control when they call another coroutine and they don't use as much memory.
- Coroutine definition starts with '`async`' and its call with '`await`'.
- '`asyncio.run(<coroutine>)`' is the main entry point for asynchronous programs.
- Functions `wait()`, `gather()` and `as_completed()` start multiple coroutines at the same time.
- `asyncio` module also provides its own `Queue`, `Event`, `Lock` and `Semaphore` classes.

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

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

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

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

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

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

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

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

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

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

Libraries

Progress Bar

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

Plot

```
# $ pip3 install matplotlib
import matplotlib.pyplot as plt
plt.plot(<x_data>, <y_data> [, label=<str>])    # Or: plt.plot(<y_data>)
plt.legend()                                         # Adds a legend.
plt.savefig(<path>)                                # Saves the figure.
plt.show()                                           # Displays the figure.
plt.clf()                                            # Clears the figure.
```

Table

Prints a CSV file as an ASCII table:

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

Curses

Runs a basic file explorer in the terminal:

```
import curses, curses.ascii, os
from curses import A_REVERSE, KEY_DOWN, KEY_UP, KEY_LEFT, KEY_RIGHT, KEY_ENTER

def main(screen):
    ch, first, selected, paths = 0, 0, 0, os.listdir()
    while ch != curses.ascii.ESC:
        height, _ = screen.getmaxyx()
        screen.erase()
        for y, filename in enumerate(paths[first : first+height]):
            screen.addstr(y, 0, filename, A_REVERSE * (selected == first + y))
        ch = screen.getch()
        selected += (ch == KEY_DOWN) - (ch == KEY_UP)
        selected = max(0, min(len(paths)-1, selected))
        first += (first <= selected - height) - (first > selected)
        if ch in [KEY_LEFT, KEY_RIGHT, KEY_ENTER, 10, 13]:
            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)
```

Logging

```
# $ pip3 install loguru
from loguru import logger

logger.add('debug_{time}.log', colorize=True) # Connects a log file.
logger.add('error_{time}.log', level='ERROR') # Another file for errors or higher.
logger.<level>('A logging message.') # Logs to file/s and prints to stderr.
```

- Levels: 'debug', 'info', 'success', 'warning', 'error', 'critical'.

Exceptions

Exception description, stack trace and values of variables are appended automatically.

```
try:
    ...
except <exception>:
    logger.exception('An error happened.')
```

Rotation

Argument that sets a condition when a new log file is created.

```
rotation=<int>|<datetime.timedelta>|<datetime.time>|<str>
```

- '<int>' - Max file size in bytes.
- '<timedelta>' - Max age of a file.
- '<time>' - Time of day.
- '<str>' - Any of above as a string: '100 MB', '1 month', 'monday at 12:00', ...

Retention

Sets a condition which old log files get deleted.

```
retention=<int>|<datetime.timedelta>|<str>
```

- '<int>' - Max number of files.
- '<timedelta>' - Max age of a file.
- '<str>' - Max age as a string: '1 week, 3 days', '2 months', ...

Scraping

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

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

WIKI_URL = 'https://en.wikipedia.org/wiki/Python_(programming_language)'
try:
    html      = requests.get(WIKI_URL).text
    document  = bs4.BeautifulSoup(html, 'html.parser')
    table     = document.find('table', class_='infobox vevent')
    python_url = table.find('th', text='Website').next_sibling.a['href']
    version   = table.find('th', text='Stable release').next_sibling.strings.__next__()
    logo_url  = table.find('img')['src']
    logo      = requests.get(f'https:{logo_url)').content
    filename  = os.path.basename(logo_url)
    with open(filename, 'wb') as file:
        file.write(logo)
    print(f'{python_url}, {version}, file:///{os.path.abspath(filename)})')
except requests.exceptions.ConnectionError:
    print("You've got problems with connection.", file=sys.stderr)
```

Web

```
# $ pip3 install bottle
from bottle import run, route, static_file, template, post, request, response
import json
```

Run

```
run(host='localhost', port=8080)           # Runs locally.
run(host='0.0.0.0', port=80)                # Runs globally.
```

Static Request

```
@route('/img/<filename>')
def send_file(filename):
    return static_file(filename, root='img_dir/')
```

Dynamic Request

```
@route('/<sport>')
def send_html(sport):
    return template('<h1>{{title}}</h1>', title=sport)
```

REST Request

```
@post('/<sport>/odds')
def send_json(sport):
    team = request.forms.get('team')
    response.headers['Content-Type'] = 'application/json'
    response.headers['Cache-Control'] = 'no-cache'
    return json.dumps({'team': team, 'odds': [2.09, 3.74, 3.68]})
```

Test:

```
# $ pip3 install requests
>>> import threading, requests
>>> threading.Thread(target=run, daemon=True).start()
>>> url = 'http://localhost:8080/football/odds'
>>> request_data = {'team': 'arsenal f.c.'}
>>> response = requests.post(url, data=request_data)
>>> response.json()
{'team': 'arsenal f.c.', 'odds': [2.09, 3.74, 3.68]}
```

Profiling

Stopwatch

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

Timing a Snippet

```
>>> from timeit import timeit
>>> timeit("''.join(str(i) for i in range(100))",
...           number=10000, globals=globals(), setup='pass')
0.34986
```

Profiling by Line

```
# $ pip3 install line_profiler memory_profiler
@profile
def main():
    a = [*range(10000)]
    b = {*range(10000)}
main()

$ kernprof -lv test.py
Line #    Hits      Time  Per Hit   % Time  Line Contents
=====
1          1      955.0    955.0     43.7    @profile
2          1      1231.0   1231.0     56.3    def main():
3          1      955.0    955.0     43.7    a = [*range(10000)]
4          1      1231.0   1231.0     56.3    b = {*range(10000)}
```

```
$ python3 -m memory_profiler test.py
Line #      Mem usage      Increment  Line Contents
=====
1      37.668 MiB      37.668 MiB  @profile
2          0.344 MiB      0.344 MiB  def main():
3      38.012 MiB      0.345 MiB  a = [*range(10000)]
4      38.477 MiB      0.465 MiB  b = {*range(10000)}
```

Call Graph

Generates a PNG image of the call graph with highlighted bottlenecks:

```
# $ pip3 install pycallgraph2; apt/brew install graphviz
import pycallgraph2 as cg, datetime

filename = f'profile-{datetime.datetime.now():%Y%m%d_%H%M%S}.png'
drawer = cg.output.GraphvizOutput(output_file=filename)
with cg.PyCallGraph(drawer):
    <code_to_be_profiled>
```

NumPy

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

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

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

<view> = <array>.reshape(<shape>)                                # Also `<array>.shape = <shape>`.
<array> = <array>.flatten()                                         # Collapses array into one dimension.
<view> = <array>.squeeze()                                         # Removes dimensions of length one.

<array> = <array>.sum/min/mean/var/std(axis)                      # Passed dimension gets aggregated.
<array> = <array>.argmin(axis)                                       # Returns indexes of smallest elements.
<array> = np.apply_along_axis(<func>, axis, <array>)           # Func can return a scalar or array.
```

- Shape is a tuple of dimension sizes. A 100x50 RGB image has shape (50, 100, 3).
- Axis is an index of the dimension that gets aggregated. Leftmost dimension has index 0. Summing the RGB image along axis 2 will return a greyscale image with shape (50, 100).
- Passing a tuple of axes will chain the operations like this: '`<array>.<method>(axis_1, keepdims=True).<method>(axis_2).squeeze()`'.

Indexing

```
<el>      = <2d_array>[row_index, column_index]                # <3d_a>[table_i, row_i, column_i]
<1d_view> = <2d_array>[row_index]                               # <3d_a>[table_i, row_i]
<1d_view> = <2d_array>[:, column_index]                         # <3d_a>[table_i, :, column_i]

<1d_array> = <2d_array>[row_indexes, column_indexes]          # <3d_a>[table_is, row_is, column_is]
<2d_array> = <2d_array>[row_indexes]                            # <3d_a>[table_is, row_is]
<2d_array> = <2d_array>[:, column_indexes]                      # <3d_a>[table_is, :, column_is]

<2d_bools> = <2d_array> >== <el>                           # <3d_array> >== <1d_array>
<1d_array> = <2d_array>[<2d_bools>]                          # <3d_array>[<2d_bools>]
```

- All examples also allow assignments.

Broadcasting

Broadcasting is a set of rules by which NumPy functions operate on arrays of different sizes and/or dimensions.

```
left  = [[0.1], [0.6], [0.8]]                                     # Shape: (3, 1)
right = [ 0.1 , 0.6 , 0.8 ]                                         # Shape: (3,)
```

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

```
left  = [[0.1], [0.6], [0.8]]                      # Shape: (3, 1)
right = [[0.1, 0.6, 0.8]]                           # Shape: (1, 3) <- !
```

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

```
left  = [[0.1, 0.1, 0.1],                                # Shape: (3, 3) <- !
         [0.6, 0.6, 0.6],
         [0.8, 0.8, 0.8]]
right = [[0.1, 0.6, 0.8],                                # Shape: (3, 3) <- !
          [0.1, 0.6, 0.8],
          [0.1, 0.6, 0.8]]
```

3. If neither non-matching dimension has size 1, raise an error.

Example

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

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

Image

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

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

<int/tuple> = <Image>.getpixel((x, y))              # Returns a pixel.
<Image>.putpixel((x, y), <int/tuple>)                # Writes a pixel to the image.
<ImagingCore> = <Image>.getdata()                   # Returns a flattened sequence of pixels.
<Image>.putdata(<list/ImagingCore>)                 # Writes a flattened sequence of pixels.
<Image>.paste(<Image>, (x, y))                      # Writes passed image to the image.

<2d_array> = np.array(<Image_L>)                   # Creates NumPy array from greyscale image.
<3d_array> = np.array(<Image_RGB/A>)               # Creates NumPy array from color image.
<Image>     = Image.fromarray(np.uint8(<array>))    # Use <array>.clip(0, 255) to clip the values.
```

Modes

- '**1**' - 1-bit pixels, black and white, stored with one pixel per byte.
- '**L**' - 8-bit pixels, greyscale.
- '**RGB**' - 3x8-bit pixels, true color.
- '**RGBA**' - 4x8-bit pixels, true color with transparency mask.
- '**HSV**' - 3x8-bit pixels, Hue, Saturation, Value color space.

Examples

Creates a PNG image of a rainbow gradient:

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

Adds noise to a PNG image:

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

Image Draw

```
from PIL import ImageDraw
<ImageDraw> = ImageDraw.Draw(<Image>)

<ImageDraw>.point((x, y))                                # Truncates floats into ints.
<ImageDraw>.line((x1, y1, x2, y2 [, ...]))            # To get anti-aliasing use Image's resize().
<ImageDraw>.arc((x1, y1, x2, y2), deg1, deg2)          # Always draws in clockwise direction.
<ImageDraw>.rectangle((x1, y1, x2, y2))                # To rotate use Image's rotate() and paste().
<ImageDraw>.polygon((x1, y1, x2, y2, ...))             # Last point gets connected to the first.
<ImageDraw>.ellipse((x1, y1, x2, y2))                  # To rotate use Image's rotate() and paste().
```

- Use '**fill=<color>**' to set the primary color.
- Use '**width=<int>**' to set the width of lines or contours.
- Use '**outline=<color>**' to set the color of the contours.
- Color can be an int, tuple, '**#rrggbbaa**' string or a color name.

Animation

Creates a GIF of a bouncing ball:

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

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

Audio

```
import wave
```

```
<Wave_read> = wave.open('<path>', 'rb')          # Opens the WAV file.  
framerate = <Wave_read>.getframerate()           # Number of frames per second.  
nchannels = <Wave_read>.getnchannels()            # Number of samples per frame.  
sampwidth = <Wave_read>.getsampwidth()             # Sample size in bytes.  
nframes = <Wave_read>.getnframes()                # Number of frames.  
<params> = <Wave_read>.getparams()                # Immutable collection of above.  
<bytes> = <Wave_read>.readframes(nframes)         # Returns next 'nframes' frames.  
  
<Wave_write> = wave.open('<path>', 'wb')          # Truncates existing file.  
<Wave_write>.setframerate(<int>)                 # 44100 for CD, 48000 for video.  
<Wave_write>.setnchannels(<int>)                  # 1 for mono, 2 for stereo.  
<Wave_write>.setsampwidth(<int>)                 # 2 for CD quality sound.  
<Wave_write>.setparams(<params>)                 # Sets all parameters.  
<Wave_write>.writeframes(<bytes>)                # Appends frames to the file.
```

- Bytes object contains a sequence of frames, each consisting of one or more samples.
- In a stereo signal, the first sample of a frame belongs to the left channel.
- Each sample consists of one or more bytes that, when converted to an integer, indicate the displacement of a speaker membrane at a given moment.
- If sample width is one byte, then the integer should be encoded unsigned.
- For all other sizes, the integer should be encoded signed with little-endian byte order.

Sample Values

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

Read Float Samples from WAV File

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

Write Float Samples to WAV File

```
def write_to_wav_file(filename, float_samples, nchannels=1, sampwidth=2, framerate=44100):  
    def get_bytes(a_float):  
        a_float = max(-1, min(1 - 2e-16, a_float))  
        a_float += sampwidth == 1  
        a_float *= pow(2, sampwidth * 8 - 1)  
        return int(a_float).to_bytes(sampwidth, 'little', signed=(sampwidth != 1))  
    with wave.open(filename, 'wb') as file:  
        file.setnchannels(nchannels)  
        file.setsampwidth(sampwidth)  
        file.setframerate(framerate)  
        file.writeframes(b''.join(get_bytes(f) for f in float_samples))
```

Examples

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

```
from math import pi, sin
samples_f = (sin(i * 2 * pi * 440 / 44100) for i in range(100000))
write_to_wav_file('test.wav', samples_f)
```

Adds noise to a mono WAV file:

```
from random import random
add_noise = lambda value: value + (random() - 0.5) * 0.03
samples_f = (add_noise(f) for f in read_wav_file('test.wav'))
write_to_wav_file('test.wav', samples_f)
```

Plays a WAV file:

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

Text to Speech

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

Synthesizer

Plays Popcorn by Gershon Kingsley:

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

F = 44100
P1 = '71J,,69J,,71J,,66J,,62J,,66J,,59J,,'
P2 = '71J,,73J,,74J,,73J,,74J,,71J,,73J,,71J,,73J,,69J,,71J,,69J,,71J,,67J,,71J,,'
get_pause = lambda seconds: it.repeat(0, int(seconds * F))
sin_f = lambda i, hz: math.sin(i * 2 * math.pi * hz / F)
get_wave = lambda hz, seconds: (sin_f(i, hz) for i in range(int(seconds * F)))
get_hz = lambda key: 8.176 * 2 ** (int(key) / 12)
parse_note = lambda note: (get_hz(note[:2]), 1/4 if 'J' in note else 1/8)
get_samples = lambda note: get_wave(*parse_note(note)) if note else get_pause(1/8)
samples_f = it.chain.from_iterable(get_samples(n) for n in f'{P1},{P1},{P2}'.split(','))
samples_b = b''.join(struct.pack('<h', int(f * 30000)) for f in samples_f)
simpleaudio.play_buffer(samples_b, 1, 2, F)
```

Pygame

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

pg.init()
screen = pg.display.set_mode((500, 500))
rect = pg.Rect(240, 240, 20, 20)
while all(event.type != pg.QUIT for event in pg.event.get()):
    deltas = {pg.K_UP: (0, -1), pg.K_RIGHT: (1, 0), pg.K_DOWN: (0, 1), pg.K_LEFT: (-1, 0)}
    for ch, is_pressed in enumerate(pg.key.get_pressed()):
        rect = rect.move(deltas[ch]) if ch in deltas and is_pressed else rect
    screen.fill((0, 0, 0))
    pg.draw.rect(screen, (255, 255, 255), rect)
    pg.display.flip()
```

Rectangle

Object for storing rectangular coordinates.

```
<Rect> = pg.Rect(x, y, width, height)           # Floats get truncated into ints.
<int> = <Rect>.x/y/centerx/centery/...          # Top, right, bottom, left. Allows assignments.
<tup.> = <Rect>.topleft/center/...              # Topright, bottomright, bottomleft. Same.
<Rect> = <Rect>.move((x, y))                   # Use move_ip() to move in-place.

<bool> = <Rect>.collidepoint((x, y))           # Checks if rectangle contains a point.
<bool> = <Rect>.colliderect(<Rect>)            # Checks if two rectangles overlap.
<int> = <Rect>.collidelist(<list_of_Rect>)      # Returns index of first colliding Rect or -1.
<list> = <Rect>.collidelistall(<list_of_Rect>)   # Returns indexes of all colliding rectangles.
```

Surface

Object for representing images.

```
<Surf> = pg.display.set_mode((width, height))      # Returns a display surface.
<Surf> = pg.Surface((width, height))                # New RGB surface. RGBA if `flags=pg.SRCALPHA`.
<Surf> = pg.image.load('<path>')                  # Loads the image. Format depends on source.
<Surf> = <Surf>.subsurface(<Rect>)                # Returns a subsurface.

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

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

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

Font

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

Sound

```
<Sound> = pg.mixer.Sound('<path>')  
<Sound>.play() # Loads the WAV file.  
# Starts playing the sound.
```

Basic Mario Brothers Example

```

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

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

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

def run(screen, images, mario, tiles):
    clock = pg.time.Clock()
    while all(event.type != pg.QUIT for event in pg.event.get()):
        keys = {pg.K_UP: D.n, pg.K_RIGHT: D.e, pg.K_DOWN: D.s, pg.K_LEFT: D.w}
        pressed = {keys.get(ch) for ch, is_prsd in enumerate(pg.key.get_pressed()) if is_prsd}
        update_speed(mario, tiles, pressed)
        update_position(mario, tiles)
        draw(screen, images, mario, tiles, pressed)
        clock.tick(28)

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

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

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

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

def draw(screen, images, mario, tiles, pressed):
    def get_marios_image_index():
        if D.s not in get_boundaries(mario.rect, tiles):
            return 4
        return next(mario.frame_cycle) if {D.w, D.e} & pressed else 6
    screen.fill((85, 168, 255))
    mario.facing_left = (D.w in pressed) if {D.w, D.e} & pressed else mario.facing_left
    screen.blit(images[get_marios_image_index() + mario.facing_left * 9], mario.rect)
    for t in tiles:
        screen.blit(images[18 if t.x in [0, (W-1)*16] or t.y in [0, (H-1)*16] else 19], t)
    pg.display.flip()

if __name__ == '__main__':
    main()

```

Pandas

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

Series

Ordered dictionary with a name.

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

```
<Sr> = Series(<list>)           # Assigns RangeIndex starting at 0.
<Sr> = Series(<dict>)           # Takes dictionary's keys for index.
<Sr> = Series(<dict/Series>, index=<list>)  # Only keeps items with keys specified in index.
```

```
<el> = <Sr>.loc[key]           # Or: <Sr>.iloc[index]
<Sr> = <Sr>.loc[keys]          # Or: <Sr>.iloc[indexes]
<Sr> = <Sr>.loc[from_key : to_key_inclusive]  # Or: <Sr>.iloc[from_i : to_i_exclusive]
```

```
<el> = <Sr>[key/index]         # Or: <Sr>.key
<Sr> = <Sr>[keys/indexes]     # Or: <Sr>[<key_range/range>]
<Sr> = <Sr>[bools]            # Or: <Sr>.i/loc[bools]
```

```
<Sr> = <Sr> ><== <el/Sr>      # Returns a Series of bools.
<Sr> = <Sr> +-* / <el/Sr>        # Items with non-matching keys get value NaN.
```

```
<Sr> = <Sr>.append(<Sr>)        # Or: pd.concat(<coll_of_Sr>)
<Sr> = <Sr>.combine_first(<Sr>)  # Adds items that are not yet present.
<Sr>.update(<Sr>)                # Updates items that are already present.
```

```
<Sr>.plot.line/area/bar/pie/hist()  # Generates a Matplotlib plot.
plt.show()                           # Displays the plot. Also plt.savefig(<path>).
```

Series – Aggregate, Transform, Map:

```
<el> = <Sr>.sum/max/mean/idxmax/all()
<Sr> = <Sr>.rank/diff/cumsum/ffill/interpl()
<Sr> = <Sr>.fillna(<el>)
```

```
# Or: <Sr>.agg(lambda <Sr>: <el>)
# Or: <Sr>.agg/transform(lambda <Sr>: <Sr>)
# Or: <Sr>.agg/transform/map(lambda <el>: <el>)
```

```
>>> sr = Series([1, 2], index=['x', 'y'])
x    1
y    2
```

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

	'rank'	['rank']	{'r': 'rank'}
sr.apply(...) sr.agg(...) sr.transform(...)	x 1 y 2	rank x 1 y 2	r x 1 y 2

- Last result has a hierarchical index. Use '`<Sr>[key_1, key_2]`' to get its values.

DataFrame

Table with labeled rows and columns.

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

<pre><DF> = DataFrame(<list_of_rows>) <DF> = DataFrame(<dict_of_columns>) <el> = <DF>.loc[row_key, column_key] <Sr/DF> = <DF>.loc[row_key/s] <Sr/DF> = <DF>.loc[:, column_key/s] <DF> = <DF>.loc[row_bools, column_bools] <Sr/DF> = <DF>[column_key/s] <DF> = <DF>[row_bools] <DF> = <DF>[<DF>of_bools>] <DF> = <DF> ><== <el/Sr/DF> <DF> = <DF> +-*/ <el/Sr/DF> <DF> = <DF>.set_index(column_key) <DF> = <DF>.reset_index() <DF> = <DF>.sort_index(ascending=True) <DF> = <DF>.sort_values(column_key/s)</pre>	<pre># Rows can be either lists, dicts or series. # Columns can be either lists, dicts or series. # Or: <DF>.iloc[row_index, column_index] # Or: <DF>.iloc[row_index/es] # Or: <DF>.iloc[:, column_index/es] # Or: <DF>.iloc[row_bools, column_bools] # Or: <DF>.column_key # Keeps rows as specified by bools. # Assigns NaN to False values. # Returns DF of bools. Sr is treated as a row. # Items with non-matching keys get value NaN. # Replaces row keys with values from a column. # Moves row keys to a column named index. # Sorts rows by row keys. # Sorts rows by the passed column/s.</pre>
--	---

DataFrame – Merge, Join, Concat:

```
>>> l = DataFrame([[1, 2], [3, 4]], index=['a', 'b'], columns=['x', 'y'])
   x  y
a  1  2
b  3  4
>>> r = DataFrame([[4, 5], [6, 7]], index=['b', 'c'], columns=['y', 'z'])
   y  z
b  4  5
c  6  7
```

	'outer'	'inner'	'left'	Description
<code>l.merge(r, on='y', how=...)</code>	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	Joins/merges on column. Also accepts left_on and right_on parameters. Uses 'inner' by default.
<code>l.join(r, lsuffix='l', rsuffix='r', how=...)</code>	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	Joins/merges on row keys. Uses 'left' by default. If r is a Series, it is treated as a column.
<code>pd.concat([l, r], axis=0, join=...)</code>	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. Use l.append(sr) to add a row instead.
<code>pd.concat([l, r], axis=1, join=...)</code>	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.
<code>l.combine_first(r)</code>	x y z a 1 2 . b 3 4 5 c . 6 7			Adds missing rows and columns. Also updates items that contain NaN. R must be a DataFrame.

DataFrame – Aggregate, Transform, Map:

```
<Sr> = <DF>.sum/max/mean/idxmax/all()          # Or: <DF>.apply/agg(lambda <Sr>: <el>)
<DF> = <DF>.rank/diff/cumsum/ffill/interpl()    # Or: <DF>.apply/agg/transfrm(lambda <Sr>: <Sr>)
<DF> = <DF>.fillna(<el>)                      # Or: <DF>.applymap(lambda <el>: <el>)
```

- All operations operate on columns by default. Pass '**axis=1**' to process the rows instead.

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

```
x y
a 1 2
b 3 4
```

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

	'rank'	['rank']	{'x': 'rank'}
<code>df.apply(...)</code> <code>df.agg(...)</code> <code>df.transform(...)</code>	x y a 1 1 b 2 2	x y rank rank a 1 1 b 2 2	a x b 1 b 2

- Use '`<DF>[col_key_1, col_key_2][row_key]`' to get the fifth result's values.

DataFrame – Plot, Encode, Decode:

```
<DF>.plot.line/bar/hist/scatter/box()
plt.show()                                     # Also: `x=column_key, y=column_key/s`.  
                                                # Displays the plot. Also plt.savefig(<path>).
```

```

<DF> = pd.read_json/html('<str/path/url>')
<DF> = pd.read_csv/pickle/excel('<path/url>')
<DF> = pd.read_sql('<table/query>', <conn.>)
<DF> = pd.read_clipboard()

<dict> = <DF>.to_dict(['d/l/s/...'])
<str> = <DF>.to_json/html/csv([<path>])
<DF>.to_pickle/excel(<path>)
<DF>.to_sql('<table_name>', <connection>)

# Returns columns as dicts, lists or series.
# Also to_markdown/latex([<path>]).
# Run `\$ pip3 install openpyxl` for xlsx files.
# Accepts SQLite3 or SQLAlchemy connection.

```

GroupBy

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

```

>>> df = DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 6]], index=list('abc'), columns=list('xyz'))
>>> df.groupby('z').get_group(6)
   x  y
b  4  5
c  7  8

```

```

<GB> = <DF>.groupby(column_key/s)
<DF> = <GB>.apply(<func>)
<GB> = <GB>[column_key]

```

Splits DF into groups based on passed column.
Maps each group. Func can return DF, Sr or el.
Single column GB. All operations return a Sr.

GroupBy – Aggregate, Transform, Map:

```

<DF> = <GB>.sum/max/mean/idxmax/all()
<DF> = <GB>.rank/diff/cumsum/ffill()
<DF> = <GB>.fillna(<el>)

# Or: <GB>.agg(lambda <Sr>: <el>)
# Or: <GB>.transform(lambda <Sr>: <Sr>)
# Or: <GB>.transform(lambda <Sr>: <Sr>)

>>> gb = df.groupby('z')
      x  y  z
3: a  1  2  3
6: b  4  5  6
     c  7  8  6

```

	'sum'	'rank'	['rank']	{'x': 'rank'}
gb.agg(...)	x y z 3 1 2 6 11 13	x y a 1 1 b 1 1 c 2 2	x y rank rank a 1 1 b 1 1 c 2 2	x a 1 b 1 c 2
gb.transform(...)	x y a 1 2 b 11 13 c 11 13	x y a 1 1 b 1 1 c 2 2		

Rolling

Object for rolling window calculations.

```

<RSR/RDF/RGB> = <Sr/DF/GB>.rolling(win_size)
<RSR/RDF/RGB> = <RDF/RGB>[column_key/s]
<Sr/DF>       = <R>.mean/sum/max()

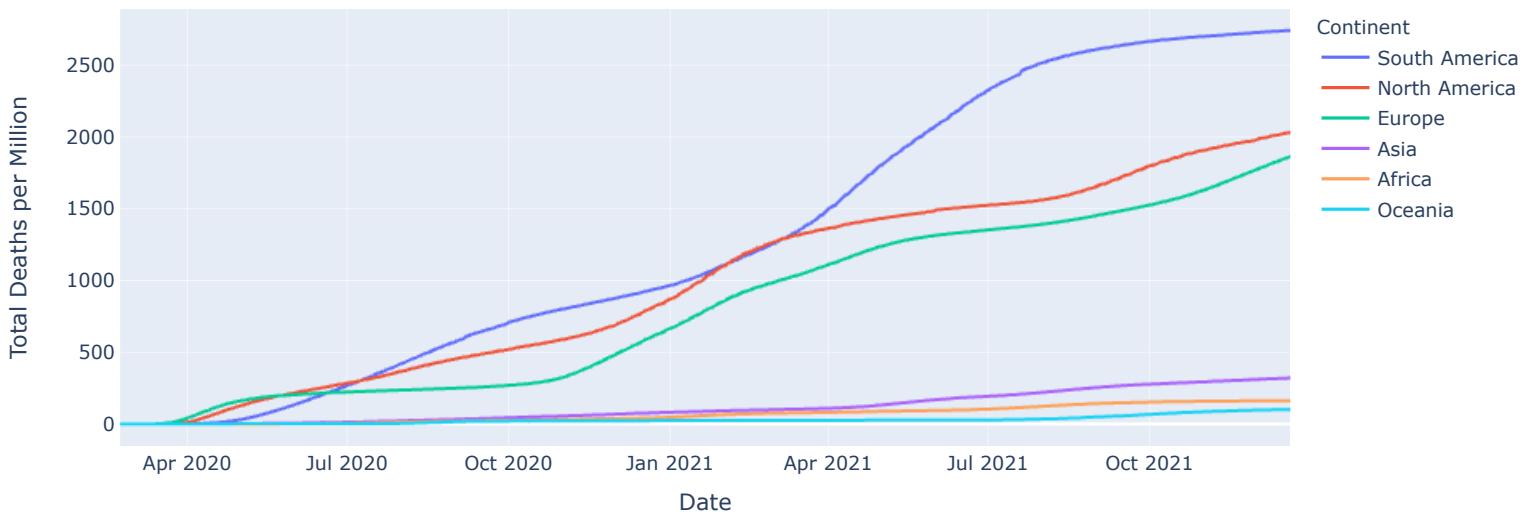
```

Also: `min_periods=None, center=False`.
Or: <RDF/RGB>.column_key
Or: <R>.apply/agg(<agg_func/str>)

Plotly

```
# $ pip3 install plotly kaleido
from plotly.express import line
<Figure> = line(<DF>, x=<col_name>, y=<col_name>) # Or: line(x=<list>, y=<list>
<Figure>.update_layout(margin=dict(t=0, r=0, b=0, l=0)) # Or: paper_bgcolor='rgba(0, 0, 0, 0)'
<Figure>.write_html/json/image('<path>') # Also: <Figure>.show()
```

Covid deaths by continent:



```
covid = pd.read_csv('https://covid.ourworldindata.org/data/owid-covid-data.csv',
                     usecols=['iso_code', 'date', 'total_deaths', 'population'])
continents = pd.read_csv('https://gist.githubusercontent.com/stevewithington/20a69c0b6d2ff',
                        '846ea5d35e5fc47f26c/raw/country-and-continent-codes-list-csv.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')
line(df, x='Date', y='Total Deaths per Million', color='Continent').show()
```

Confirmed covid cases, Dow Jones, Gold, and Bitcoin price:



```

import pandas as pd
import plotly.graph_objects as go

def main():
    display_data(wrangle_data(*scrape_data()))

def scrape_data():
    def scrape_covid():
        url = 'https://covid.ourworldindata.org/data/owid-covid-data.csv'
        df = pd.read_csv(url, usecols=['location', 'date', 'total_cases'])
        return df[df.location == 'World'].set_index('date').total_cases
    def scrape_yahoo(slug):
        url = f'https://query1.finance.yahoo.com/v7/finance/download/{slug}' + \
            '?period1=1579651200&period2=9999999999&interval=1d&events=history'
        df = pd.read_csv(url, usecols=['Date', 'Close'])
        return df.set_index('Date').Close
    out = scrape_covid(), scrape_yahoo('BTC-USD'), scrape_yahoo('GC=F'), scrape_yahoo('^DJI')
    return map(pd.Series.rename, out, ['Total Cases', 'Bitcoin', 'Gold', 'Dow Jones'])

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

def display_data(df):
    figure = go.Figure()
    for col_name in reversed(df.columns):
        yaxis = 'y1' if col_name == 'Total Cases' else 'y2'
        trace = go.Scatter(x=df.index, y=df[col_name], name=col_name, yaxis=yaxis)
        figure.add_trace(trace)
    figure.update_layout(
        yaxis1=dict(title='Total Cases', rangemode='tozero'),
        yaxis2=dict(title='%', rangemode='tozero', overlaying='y', side='right'),
        legend=dict(x=1.1),
        height=450
    ).show()

if __name__ == '__main__':
    main()

```

PySimpleGUI

```
# $ pip3 install PySimpleGUI
import PySimpleGUI as sg
layout = [[sg.Text("What's your name?")], [sg.Input()], [sg.Button('Ok')]]
window = sg.Window('Window Title', layout)
event, values = window.read()
print(f'Hello {values[0]}!' if event == 'Ok' else '')
```

Appendix

Cython

Library that compiles Python code into C.

```
# $ pip3 install cython
import pyximport; pyximport.install()
import <cython_script>
<cython_script>.main()
```

Definitions:

- All '**cdef**' definitions are optional, but they contribute to the speed-up.
- Script needs to be saved with a '**pyx**' extension.

```
cdef <ctype> <var_name> = <el>
cdef <ctype>[n_elements] <var_name> = [<el_1>, <el_2>, ...]
cdef <ctype/void> <func_name>(<ctype> <arg_name>): ...

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

cdef enum <enum_name>: <member_name_1>, <member_name_2>, ...
```

PyInstaller

```
$ pip3 install pyinstaller
$ pyinstaller script.py                      # Compiles into './dist/script' directory.
$ pyinstaller script.py --onefile            # Compiles into './dist/script' console app.
$ pyinstaller script.py --windowed          # Compiles into './dist/script' windowed app.
$ pyinstaller script.py --add-data '<path>::' # Adds file to the root of the executable.
```

- File paths need to be updated to '**os.path.join(sys._MEIPASS, <path>)**'.

Basic Script Template

```
#!/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

- Only available in the PDF.
- Ctrl+F / ⌘F is usually sufficient.
- Searching '#<title>' will limit the search to the titles.