March 14, 2018 / Jure Šorn

Comprehensive Python Cheatsheet

Download text file, PDF, Fork me on GitHub or Check out FAQ.

# 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, Closure, Decorator, Class, Duck_Types, Enum, Exceptions],
    '4. System':      [Print, Input, Command_Line_Arguments, Open, Path, Command_Execution],
    '5. Data':        [JSON, Pickle, CSV, SQLite, Bytes, Struct, Array, MemoryView, Deque],
    '6. Advanced':    [Threading, Operator, Introspection, Metaprograming, Eval, Coroutine],
    '7. Libraries':   [Progress_Bar, Plot, Table, Curses, Logging, Scraping, Web, Profile,
                       NumPy, Image, Animation, Audio, Synthesizer]
}
```

# Main

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

# List

```
<list> = <list>[from_inclusive : to_exclusive : ±step_size]
```

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

```
<list>.sort()
<list>.reverse()
<list> = sorted(<collection>)
<iter> = reversed(<list>)
```

```
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, x: out * x, <collection>)
list_of_chars    = list(<str>)
```

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

```
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>)        # Creates a dict with default value of type.
<dict> = collections.defaultdict(lambda: 1)     # Creates a dict with default value 1.
```

```
<dict>.update(<dict>)
<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.
```

```
value = <dict>.pop(key)                         # Removes item or raises KeyError.
{k: v for k, v in <dict>.items() if k in keys}  # Filters dictionary by keys.
```

Counter

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

# Set

```
<set> = set()
```

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

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

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

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> = ()
<tuple> = (<el>, )
<tuple> = (<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
>>> p._fields  # Or: Point._fields
('x', 'y')
```

# Range

```
<range> = range(to_exclusive)
<range> = range(from_inclusive, to_exclusive)
<range> = range(from_inclusive, to_exclusive, ±step_size)
```

```
from_inclusive = <range>.start
to_exclusive   = <range>.stop
```

# Enumerate

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

# Iterator

```
<iter> = iter(<collection>)                 # `iter(<iter>)` returns unmodified iterator.
<iter> = iter(<function>, to_exclusive)     # Sequence of return values until 'to_exclusive'.
<el>   = next(<iter> [, default])           # Raises StopIteration or returns 'default' on end.
```

Itertools

```
from itertools import count, repeat, cycle, chain, islice
```

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

```
<iter> = chain(<coll_1>, <coll_2> [, ...])  # Empties collections in order.
<iter> = chain.from_iterable(<collection>)  # Empties collections inside a collection in order.
```

```
<iter> = islice(<collection>, to_exclusive)
<iter> = islice(<collection>, from_inclusive, to_exclusive)
<iter> = islice(<collection>, from_inclusive, to_exclusive, +step_size)
```

# 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
```

ABC

An abstract base class introduces virtual subclasses, that don't inherit from it but are still recognized by isinstance() and issubclass().

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

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

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

	Integral	Rational	Real	Complex	Number
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)    # Splits on line breaks. Keeps them if 'keepends'.
<str>  = <str>.join(<coll_of_strings>)       # Joins elements using string as 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 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.
<bool> = <str>.isnumeric()                   # True if str contains only numeric characters.
<list> = textwrap.wrap(<str>, width)         # Nicely breaks string into lines.
```

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

Char

```
<str> = chr(<int>)                           # Converts int to unicode char.
<int> = ord(<str>)                           # Converts unicode char to int.
```

```
>>> ord('0'), ord('9')
(48, 57)
>>> ord('A'), ord('Z')
(65, 90)
>>> ord('a'), ord('z')
(97, 122)
```

# Regex

```
import re
<str>   = re.sub(<regex>, new, text, count=0)  # Substitutes all occurrences.
<list>  = re.findall(<regex>, text)            # Returns all occurrences.
<list>  = re.split(<regex>, text, maxsplit=0)  # Use brackets in regex to keep the matches.
<Match> = re.search(<regex>, text)             # Searches for first occurrence of 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.
```

- **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 newline.**
- **Use `r'\1'` or `'\\1'` for backreference.**
- **Add `'?'` after an operator to make it non-greedy.**

Match Object

```
<str>   = <Match>.group()                      # Whole match. Also group(0).
<str>   = <Match>.group(1)                     # Part in first bracket.
<tuple> = <Match>.groups()                     # All bracketed parts.
<int>   = <Match>.start()                      # Start index of a match.
<int>   = <Match>.end()                        # Exclusive end index of a match.
```

Special Sequences

- By default digits, whitespaces and alphanumerics from all alphabets are matched, unless `'flags=re.ASCII'` argument is used.
- Use capital letters for negation.

```
'\d' == '[0-9]'                                # Digit
'\s' == '[ \t\n\r\f\v]'                        # Whitespace
'\w' == '[a-zA-Z0-9_]'                         # Alphanumeric
```

# Format

```
<str> = f'{<el_1>}, {<el_2>}'
<str> = '{}, {}'.format(<el_1>, <el_2>)
```

Attributes

```
>>> from collections import namedtuple
>>> Person = 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>'
```

Strings

`'!r'` calls object's repr() method, instead of str(), to get a string.

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

Numbers

```
{ 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 float 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%'
567.89	'567.89'	'567.890000'	'5.678900e+02'	'56789.000000%'

	{<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%'
567.89	'5.7e+02'	'567.89'	'5.68e+02'	'56789.00%'

Ints

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

# Numbers

Types

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

- **`'int(<str>)'` and `'float(<str>)'` raise ValueError on malformed strings.**
- **Decimal numbers can be represented exactly, unlike floats where `'1.1 + 2.2 != 3.3'`.**
- **Their precision can be adjusted with `'decimal.getcontext().prec = <int>'`.**

Basic Functions

```
<num> = pow(<num>, <num>)              # Or: <num> ** <num>
<num> = abs(<num>)
<int> = round(<num>)
<num> = round(<num>, ±ndigits)         # `round(126, -1) == 130`
```

Math

```
from math import e, pi, inf, nan
from math import cos, acos, sin, asin, tan, atan, degrees, radians
from math import log, log10, log2
```

Statistics

```
from statistics import mean, median, variance, pvariance, pstdev
```

Random

```
from random import random, randint, choice, shuffle
<float> = random()
<int>   = randint(from_inclusive, to_inclusive)
<el>    = choice(<list>)
shuffle(<list>)
```

Bin, Hex

```
<int>     = 0b<bin>                  # Or: 0x<hex>
<int>     = int('<bin>', 2)          # Or: int('<hex>', 16)
<int>     = int('0b<bin>', 0)        # Or: int('0x<hex>', 0)
'0b<bin>' = bin(<int>)               # Or: '0x<hex>' = hex(<int>)
```

Bitwise Operators

```
<int>     = <int> & <int>            # And
<int>     = <int> | <int>            # Or
<int>     = <int> ^ <int>            # Xor (0 if both bits equal)
<int>     = <int> << n_bits          # Shift left
<int>     = <int> >> n_bits          # Shift right
<int>     = ~<int>                   # Compliment (flips bits)
```

# Combinatorics

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

```
from itertools import product, combinations, combinations_with_replacement, permutations
```

```
>>> 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)]
```

```
>>> product('ab', '12')
[('a', '1'), ('a', '2'),
 ('b', '1'), ('b', '2')]
```

```
>>> combinations('abc', 2)
[('a', 'b'), ('a', 'c'), ('b', 'c')]
```

```
>>> combinations_with_replacement('abc', 2)
[('a', 'a'), ('a', 'b'), ('a', 'c'),
 ('b', 'b'), ('b', 'c'),
 ('c', 'c')]
```

```
>>> permutations('abc', 2)
[('a', 'b'), ('a', 'c'),
 ('b', 'a'), ('b', 'c'),
 ('c', 'a'), ('c', 'b')]
```

# Datetime

- **Module 'datetime' provides 'date' <D>, 'time' <T>, 'datetime' <DT> and 'timedelta' <TD> classes. All are immutable and hashable.**
- **Time and datetime 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
```

Constructors

```
<D>  = date(year, month, day)
<T>  = time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None, fold=0)
<DT> = datetime(year, month, day, hour=0, minute=0, second=0, ...)
<TD> = timedelta(days=0, seconds=0, microseconds=0, milliseconds=0,
                 minutes=0, hours=0, weeks=0)
```

- Use `'<D/DT>.weekday()'` to get the day of the week (Mon == 0).
- `'fold=1'` means second pass in case of time jumping back for one hour.

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('<Cont.>/<City>')          # 'Continent/City_Name' timezone or None.
```

```
<DTa>    = <DT>.astimezone(<tzinfo>)        # Datetime, converted to passed timezone.
<Ta/DTa> = <T/DT>.replace(tzinfo=<tzinfo>)  # Unconverted object with 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 Christ, at midnight.
<DTn>    = DT.fromtimestamp(<real>)         # Local time DTn from seconds since Epoch.
<DTa>    = DT.fromtimestamp(<real>, <tz.>)  # Aware datetime from seconds since Epoch.
```

- ISO strings come in following forms: `'YYYY-MM-DD'`, `'HH:MM:SS.ffffff[±<offset>]'`, or both separated by a space or a `'T'`. Offset is formatted as: `'HH:MM'`.
- On Unix systems Epoch is `'1970-01-01 00:00 UTC'`, `'1970-01-01 01:00 CET'`, ...

Decode

```
<str>    = <D/T/DT>.isoformat()             # ISO string representation.
<str>    = <D/T/DT>.strftime('<format>')    # Custom string representation.
<int>    = <D/DT>.toordinal()               # Days since Christ, ignoring time and tz.
<float>  = <DTn>.timestamp()                # Seconds since Epoch from DTn in local time.
<float>  = <DTa>.timestamp()                # Seconds since Epoch from DTa.
```

Format

```
>>> from datetime import datetime
>>> dt = datetime.strptime('2015-05-14 23:39:00.00 +0200', '%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"
```

- When parsing, `'%z'` also accepts `'±HH:MM'`.
- For abbreviated weekday and month use `'%a'` and `'%b'`.

Arithmetics

```
<TD>     = <D/DT> -  <D/DT>
<D/DT>   = <D/DT> ±  <TD>
<TD>     = <TD>   ±  <TD>
<TD>     = <TD>   */ <real>
```

# Arguments

Inside Function Call

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

Inside Function Definition

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

# 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(x, y, z):                # 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, 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)
```

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

Other Uses

```
<list>  = [*<collection> [, ...]]
<set>   = {*<collection> [, ...]}
<tuple> = (*<collection>, [...])
<dict>  = {**<dict> [, ...]}
```

```
head, *body, tail = <collection>
```

# Inline

Lambda

```
<function> = lambda: <return_value>
<function> = lambda <argument_1>, <argument_2>: <return_value>
```

Comprehension

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

```
out = [i+j for i in range(10) for j in range(10)]
```

Is the same as:

```
out = []
for i in range(10):
    for j in range(10):
        out.append(i+j)
```

Map, Filter, Reduce

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

Any, All

```
<bool> = any(<collection>)                          # False if empty.
<bool> = all(el[1] for el in <collection>)          # True if empty.
```

If - Else

```
<expression_if_true> if <condition> else <expression_if_false>
```

```
>>> [a if a else 'zero' for a in (0, 1, 0, 3)]
['zero', 1, 'zero', 3]
```

Namedtuple, Enum, Dataclass

```
from collections import namedtuple
Point     = namedtuple('Point', 'x y')
point     = Point(0, 0)
```

```
from enum import Enum
Direction = Enum('Direction', 'n e s w')
direction = Direction.n
```

```
from dataclasses import make_dataclass
Creature  = make_dataclass('Creature', ['location', 'direction'])
creature  = Creature(Point(0, 0), Direction.n)
```

# Closure

We have 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 a 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>)'`.

Nonlocal

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.

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

Debugger Example

Decorator that prints function's name every time it gets called.

```python
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 metadata of function func() to function out().
- Without it `'add.__name__'` would return `'out'`.

LRU Cache

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

```python
from functools import lru_cache

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

- Recursion depth is limited 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.

```python
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
```

# 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().

Str() use cases:

```
print(<el>)
print(f'{<el>}')
raise Exception(<el>)
loguru.logger.debug(<el>)
csv.writer(<file>).writerow([<el>])
```

Repr() use cases:

```
print([<el>])
print(f'{<el>!r}')
>>> <el>
loguru.logger.exception()
Z = dataclasses.make_dataclass('Z', ['a']); print(Z(<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:

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

Property

```
class MyClass:
    @property
    def a(self):
        return self._a

    @a.setter
    def a(self, value):
        self._a = value
```

```
>>> el = MyClass()
>>> el.a = 123
>>> el.a
123
```

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'` or immutable and hashable with `'frozen=True'`.
- Function field() is needed because `'<attr_name>: list = []'` would make a list that is shared among all instances.
- Default_factory can be any callable.

Inline:

```
from dataclasses import make_dataclass
<class> = make_dataclass('<class_name>', <list_of_attribute_names>)
```

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.**

```python
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().**

```python
class MyHashable:
    def __init__(self, a):
        self._a = copy.deepcopy(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.**

```python
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

- Next() should return next item or raise StopIteration.
- Iter() should return 'self'.

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

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

Callable

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

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

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

Context Manager

- Enter() should lock the resources and return an object.
- Exit() should release the resources.

```python
class MyOpen():
    def __init__(self, filename):
        self.filename = filename
    def __enter__(self):
        self.file = open(self.filename)
        return self.file
    def __exit__(self, *args):
        self.file.close()
```

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

List of covered context managers:

```python
with open('<path>') as file: ...
with wave.open('<path>') as wave_file: ...
with memoryview(<bytes/bytearray/array>) as view: ...
with concurrent.futures.ThreadPoolExecutor() as executor: ...
db = sqlite3.connect('<path>'); with db: ...
lock = threading.RLock(); with lock: ...
```

# 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):
        for el in self.a:
            yield el
```

```
>>> z = MyIterable([1, 2, 3])
>>> iter(z)
<generator object MyIterable.__iter__ at 0x1026c18b8>
>>> 1 in z
True
```

Collection

- Only required methods are iter() and len().
- 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'.

```
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 index or raise IndexError.
- Iter() and contains() automatically work on any object that has getitem() defined.
- Reversed() automatically works on any object that has getitem() and len() defined.

```
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)
```

Collections.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, collections.abc.Sequence)'` would return False even if MySequence had all the methods defined.**

```python
class MyAbcSequence(collections.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 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

```python
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()

    @classmethod
    def get_member_names(cls):
        return [a.name for a in cls.__members__.values()]
```

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

```python
<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.
name     = <member>.name
value    = <member>.value
```

```python
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>))
```

Inline

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

Functions can not 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)})
```

- **Another solution in this particular case, is to use `'and_'` and `'or_'` functions from module operator.**

# Exceptions

Basic Example

```
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>
```

Catching Exceptions

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

- **Also catches subclasses of the exception.**

Raising Exceptions

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

Useful built-in exceptions:

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

Re-raising caught exception:

```
except <exception>:
    <code>
    raise
```

Common Built-in Exceptions

```
BaseException
 ├── SystemExit                   # Raised by the sys.exit() function.
 ├── KeyboardInterrupt            # Raised when the user hits the interrupt key.
 └── Exception                    # User-defined exceptions should be derived from this class.
      ├── StopIteration           # Raised by next() when run on an empty iterator.
      ├── ArithmeticError         # Base class for arithmetic errors.
      │    └── ZeroDivisionError  # Raised when dividing by zero.
      ├── AttributeError          # Raised when an attribute is missing.
      ├── EOFError                # Raised by input() when it hits end-of-file condition.
      ├── LookupError             # Raised when a look-up on a sequence or dict fails.
      │    ├── IndexError         # Raised when a sequence index is out of range.
      │    └── KeyError           # Raised when a dictionary key is not found.
      ├── NameError               # Raised when a variable name is not found.
      ├── OSError                 # Failures such as “file not found” or “disk full”.
      │    └── FileNotFoundError  # When a file or directory is requested but doesn't exist.
      ├── RuntimeError            # Raised by errors that don't fall in other categories.
      │    └── RecursionError     # Raised when the the maximum recursion depth is exceeded.
      ├── TypeError               # Raised when an argument is of wrong type.
      └── ValueError              # When an argument is of right type but inappropriate value.
           └── UnicodeError       # Raised when encoding/decoding strings from/to bytes fails.
```

Collections and their exceptions:

	list	dict	set
getitem()	IndexError	KeyError	
pop()	IndexError	KeyError	KeyError
remove()	ValueError		KeyError
index()	ValueError		

User-defined Exceptions

```
class MyError(Exception):
    pass

class MyInputError(MyError):
    pass
```

# Print

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

- Use `'file=sys.stderr'` for errors.
- Use `'flush=True'` to forcibly flush the stream.

Pretty Print

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

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

# Input

- Reads a line from user input or pipe if present.
- Trailing newline gets stripped.
- Prompt string is printed to the standard output before reading input.
- Raises EOFError when user hits EOF or input stream gets exhausted.

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

# Argv

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

Argparse

```
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.
- Use `'default=<el>'` to set the default value.
- Use `'type=FileType(<mode>)'` for files.

# Open

Opens a file and returns a corresponding file object.

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

- `'encoding=None'` means 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 either '\n', '\r' or '\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.

Exceptions

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

File

```
<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 pos., 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 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(<coll.>)          # Writes a coll. of strings or bytes objects.
<file>.flush()                      # Flushes write buffer.
```

- 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)
```

# Path

```
from os import path, listdir
from glob import glob
```

```
<bool> = path.exists('<path>')
<bool> = path.isfile('<path>')
<bool> = path.isdir('<path>')
```

```
<list> = listdir('<path>')        # List of filenames located at path.
<list> = glob('<pattern>')        # Filenames matching the wildcard pattern.
```

Pathlib

```python
from pathlib import Path
```

```python
cwd    = Path()
<Path> = Path('<path>' [, '<path>', <Path>, ...])
<Path> = <Path> / '<dir>' / '<file>'
```

```python
<bool> = <Path>.exists()
<bool> = <Path>.is_file()
<bool> = <Path>.is_dir()
```

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

```python
<str>  = str(<Path>)                # Path as a string.
<str>  = <Path>.name                # Final component.
<str>  = <Path>.stem                # Final component without extension.
<str>  = <Path>.suffix              # Final component's extension.
<tup.> = <Path>.parts               # All components as strings.
```

```python
<Path> = <Path>.resolve()           # Returns absolute path without symlinks.
<Path> = <Path>.parent              # Returns path without final component.
<file> = open(<Path>)               # Opens the file and returns a file object.
```

# OS Commands

Files and Directories

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

```python
import os, shutil
```

```python
os.chdir(<path>)                    # Changes current working directory.
os.mkdir(<path>, mode=0o777)        # Creates a directory.
```

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

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

```python
shutil.copy(from, to)               # Copies the file.
shutil.copytree(from, to)           # Copies the entire directory tree.
```

```python
<str>  = os.getcwd()                # Returns the current working directory.
<iter> = os.scandir(path='.')       # Returns os.DirEntry objects located at path.
```

DirEntry:

```python
<bool> = <DirEntry>.is_file()
<bool> = <DirEntry>.is_dir()
```

```python
<str>  = <DirEntry>.path            # Path as a string.
<str>  = <DirEntry>.name            # Final component.
```

```python
<Path> = Path(<DirEntry>)           # Path object.
<file> = open(<DirEntry>)           # File object.
```

Shell Commands

```
import os
<str> = os.popen('<shell_command>').read()
```

Using subprocess:

```
>>> import subprocess, shlex
>>> a = subprocess.run(shlex.split('ls -a'), stdout=subprocess.PIPE)
>>> a.stdout
b'.\n..\nfile1.txt\nfile2.txt\n'
>>> a.returncode
0
```

# JSON

```
import json
<str>    = json.dumps(<object>, ensure_ascii=True, indent=None)
<object> = json.loads(<str>)
```

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

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

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.

```python
import csv
```

Read

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

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

Write

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

- **File must be opened with `'newline=""'` argument, or an extra '\r' will be added on platforms that use '\r\n' linendings!**

Parameters

- **`'dialect'` - Master parameter that sets the default values.**
- **`'delimiter'` - A one-character string used to separate fields.**
- **`'quotechar'` - Character for quoting fields that contain special characters.**
- **`'doublequote'` - Whether quotechars inside fields get doubled or escaped.**
- **`'skipinitialspace'` - Whether whitespace after delimiter gets stripped.**
- **`'lineterminator'` - How does writer terminate lines.**
- **`'quoting'` - Controls the amount of quoting: 0 - as necessary, 1 - all.**
- **`'escapechar'` - Character for escaping 'quotechar' if 'doublequote' is false.**

Dialects

	excel	excel_tab	unix_dialect
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

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

Write Rows to CSV File

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

# SQLite

Server-less database engine that stores each database into separate file.

```
import sqlite3
db = sqlite3.connect('<path>')                  # Also ':memory:'.
...
db.close()
```

- **New database will be created if path doesn't exist.**

Read

```
<cursor> = db.execute('<query>')                # Can raise sqlite3.OperationalError.
<tuple>  = <cursor>.fetchone()                  # Returns next row. Also next(<cursor>).
<list>   = <cursor>.fetchall()                  # Returns remaining rows.
```

- **Returned values can be of type str, int, float, bytes or None.**

Write

```
db.execute('<query>')
db.commit()
```

Or:

```
with db:
    db.execute('<query>')
```

Placeholders

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

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

Example

```
>>> db = sqlite3.connect('test.db')
>>> db.execute('create table t (a, b, c)')
>>> db.execute('insert into t values (1, 2, 3)')
>>> db.execute('select * from t').fetchall()
[(1, 2, 3)]
```

- **In this example values are not actually saved because `'db.commit()'` was omitted.**

MySQL

Has a very similar interface, with differences listed below.

```
# $ pip3 install mysql-connector
from mysql import connector
db = connector.connect(host=<str>, user=<str>, password=<str>, database=<str>)
<cursor> = db.cursor()
<cursor>.execute('<query>')                     # Only cursor has execute method.
<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>'                       # Only accepts ASCII characters and \x00 - \xff.
<int>   = <bytes>[<index>]               # Returns int in range from 0 to 255.
<bytes> = <bytes>[<slice>]               # Returns bytes even if it has only one element.
<bytes> = <bytes>.join(<coll_of_bytes>)  # Joins elements using bytes object as separator.
```

Encode

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

Decode

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

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.**
- **Machine’s native type sizes and byte order are used by default.**

```
from struct import pack, unpack, iter_unpack
<bytes>  = pack('<format>', <num_1> [, <num_2>, ...])
<tuple>  = unpack('<format>', <bytes>)
<tuples> = iter_unpack('<format>', <bytes>)
```

Example

```
>>> 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 sizes start format string with:

- `'='` - native byte order
- `'<'` - little-endian
- `'>'` - big-endian

Integer types. Use capital letter for unsigned type. Standard sizes are in brackets:

- `'x'` - pad byte
- `'b'` - char (1)
- `'h'` - short (2)
- `'i'` - int (4)
- `'l'` - long (4)
- `'q'` - long long (8)

Floating point types:

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

Array

List that can only hold numbers of a predefined type. Available types and their sizes in bytes are listed above.

```
from array import array
<array> = array('<typecode>', <collection>)    # Array from coll. of numbers.
<array> = array('<typecode>', <bytes>)         # Array from bytes object.
<bytes> = <array>.tobytes()
```

# 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.

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

```
<bin_file>.write(<mview>)                      # Writes mview to a binary file.
<bytes> = bytes(<mview>)                       # Creates a new bytes object.
<bytes> = <bytes>.join(<coll_of_mviews>)       # Joins mviews using bytes object as sep.
```

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

# 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.
<el> = <deque>.popleft()                       # Raises IndexError if empty.
<deque>.extendleft(<collection>)               # Collection gets reversed.
<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 there is an I/O operation in the thread.

```
from threading import Thread, RLock
```

### Thread

```
thread = Thread(target=<function>, args=(<first_arg>, ))
thread.start()
...
<bool> = thread.is_alive()   # Checks if thread has finished executing.
thread.join()                # Waits for thread to finish.
```

Lock

```
lock = RLock()
lock.acquire()               # Waits for lock to be available.
...
lock.release()
```

Or:

```
lock = RLock()
with lock:
    ...
```

Thread Pool

```
from concurrent.futures import ThreadPoolExecutor
with ThreadPoolExecutor(max_workers=None) as executor:
    <iter>   = executor.map(lambda x: x + 1, range(3))         # (1, 2, 3)
    <iter>   = executor.map(lambda x, y: x + y, 'abc', '123')  # ('a1', 'b2', 'c3')
    <Future> = executor.submit(<function> [, <arg_1>, ...])
```

```
<bool> = <Future>.done()     # Checks if thread has finished executing.
<obj>  = <Future>.result()   # 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

```python
from operator import add, sub, mul, truediv, floordiv, mod, pow, neg, abs
from operator import eq, ne, lt, le, gt, ge
from operator import and_, or_, not_
from operator import itemgetter, attrgetter, methodcaller
```

```python
import operator as op
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>)
LogicOp          = enum.Enum('LogicOp', {'AND': op.and_, 'OR' : op.or_})
last_el          = op.methodcaller('pop')(<list>)
```

# Introspection

Inspecting code at runtime.

Variables

```python
<list> = dir()      # Names of variables in current scope.
<dict> = locals()   # Dict of local variables. Also vars().
<dict> = globals()  # Dict of global variables.
```

Attributes

```python
<dict> = vars(<object>)
<bool> = hasattr(<object>, '<attr_name>')
value  = getattr(<object>, '<attr_name>')
setattr(<object>, '<attr_name>', value)
```

Parameters

```python
from inspect import signature
<sig>        = signature(<function>)
no_of_params = len(<sig>.parameters)
param_names  = list(<sig>.parameters.keys())
```

# Metaprograming

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.

```python
<class> = type(<class_name>, <parents_tuple>, <attributes_dict>)
```

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

Meta Class

Class that creates classes.

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

Or:

```python
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 class of returned instance (MyMetaClass 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)`), in which case init() is not called.

Metaclass Attribute

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

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

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

Type Diagram

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

```text
┏━━━━━━━━━┯━━━━━━━━━━━━━┓
┃ Classes │ Metaclasses ┃
┠─────────┼─────────────┨
┃ MyClass → MyMetaClass ┃
┃         │     ↓       ┃
┃  object ───→ type ←╮  ┃
┃         │    ↑ ╰───╯  ┃
┃   str ───────╯        ┃
┗━━━━━━━━━┷━━━━━━━━━━━━━┛
```

Inheritance Diagram

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

```text
┏━━━━━━━━━┯━━━━━━━━━━━━━┓
┃ Classes │ Metaclasses ┃
┠─────────┼─────────────┨
┃ MyClass │ MyMetaClass ┃
┃    ↓    │     ↓       ┃
┃  object ←──── type    ┃
┃    ↑    │             ┃
┃   str   │             ┃
┗━━━━━━━━━┷━━━━━━━━━━━━━┛
```

# Eval

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

# Coroutine

- Any function that contains a `'(yield)'` expression returns a coroutine.
- Coroutines are similar to iterators, but data needs to be pulled out of an iterator by calling `'next(<iter>)'`, while we push data into the coroutine by calling `'<coroutine>.send(<el>)'`.
- Coroutines provide more powerful data routing possibilities than iterators.

Helper Decorator

- All coroutines must first be "primed" by calling `'next(<coroutine>)'`.
- Remembering to call next() is easy to forget.
- Solved by wrapping functions that return a coroutine with a decorator:

```python
def coroutine(func):
    def out(*args, **kwargs):
        cr = func(*args, **kwargs)
        next(cr)
        return cr
    return out
```

Pipeline Example

```python
def reader(target):
    for i in range(10):
        target.send(i)
    target.close()

@coroutine
def adder(target):
    while True:
        value = (yield)
        target.send(value + 100)

@coroutine
def printer():
    while True:
        value = (yield)
        print(value)

reader(adder(printer()))  # 100, 101, ..., 109
```

Libraries

# Progress Bar

```
# $ pip3 install tqdm
from tqdm import tqdm
from time import sleep
for el in tqdm([1, 2, 3]):
    sleep(0.2)
```

# Plot

```
# $ pip3 install matplotlib
from matplotlib import pyplot
pyplot.plot(<y_data> [, label=<str>])
pyplot.plot(<x_data>, <y_data>)
pyplot.legend()                                # Adds a legend.
pyplot.savefig(<filename>)                     # Saves figure.
pyplot.show()                                  # Displays figure.
pyplot.clf()                                   # Clears 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 = [a.title() for a in next(rows)]
    table  = tabulate.tabulate(rows, header)
    print(table)
```

# Curses

Clears the terminal, prints a message and waits for an ESC key press:

```
from curses import wrapper, curs_set, ascii
from curses import KEY_UP, KEY_RIGHT, KEY_DOWN, KEY_LEFT

def main():
    wrapper(draw)

def draw(screen):
    curs_set(0)             # Makes cursor invisible.
    screen.nodelay(True)    # Makes getch() non-blocking.
    screen.clear()
    screen.addstr(0, 0, 'Press ESC to quit.')
    while screen.getch() != ascii.ESC:
        pass

def get_border(screen):
    from collections import namedtuple
    P = namedtuple('P', 'x y')
    height, width = screen.getmaxyx()
    return P(width-1, height-1)

if __name__ == '__main__':
    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.')
```

- 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 Wikipedia page:

```
# $ pip3 install requests beautifulsoup4
import requests
from bs4 import BeautifulSoup
url   = 'https://en.wikipedia.org/wiki/Python_(programming_language)'
html  = requests.get(url).text
doc   = BeautifulSoup(html, 'html.parser')
table = doc.find('table', class_='infobox vevent')
rows  = table.find_all('tr')
link  = rows[11].find('a')['href']
ver   = rows[6].find('div').text.split()[0]
url_i = rows[0].find('img')['src']
image = requests.get(f'https:{url_i}').content
with open('test.png', 'wb') as file:
    file.write(image)
print(link, ver)
```

# Web

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

Run

```
run(host='localhost', port=8080)
run(host='0.0.0.0', port=80, server='cherrypy')
```

Static Request

```
@route('/img/<image>')
def send_image(image):
    return static_file(image, 'images/', mimetype='image/png')
```

Dynamic Request

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

REST Request

```
@post('/odds/<sport>')
def odds_handler(sport):
    team = request.forms.get('team')
    home_odds, away_odds = 2.44, 3.29
    response.headers['Content-Type'] = 'application/json'
    response.headers['Cache-Control'] = 'no-cache'
    return json.dumps([team, home_odds, away_odds])
```

Test:

```
# $ pip3 install requests
>>> import requests
>>> url  = 'http://localhost:8080/odds/football'
>>> data = {'team': 'arsenal f.c.'}
>>> response = requests.post(url, data=data)
>>> response.json()
['arsenal f.c.', 2.44, 3.29]
```

# Profiling

Stopwatch

```
from time import time
start_time = time()                     # Seconds since the Epoch.
...
duration = time() - start_time
```

High performance:

```
from time import perf_counter
start_time = perf_counter()             # Seconds since restart.
...
duration = perf_counter() - start_time
```

Timing a Snippet

```
>>> from timeit import timeit
>>> timeit('"-".join(str(a) for a 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                                    @profile
     2                                    def main():
     3      1   1128.0   1128.0     27.4      a = [*range(10000)]
     4      1   2994.0   2994.0     72.6      b = {*range(10000)}
```

```
$ python3 -m memory_profiler test.py
Line #         Mem usage      Increment   Line Contents
=======================================================
     1        35.387 MiB     35.387 MiB   @profile
     2                                    def main():
     3        35.734 MiB      0.348 MiB       a = [*range(10000)]
     4        36.160 MiB      0.426 MiB       b = {*range(10000)}
```

Call Graph

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

```
# $ pip3 install pycallgraph
from pycallgraph import output, PyCallGraph
from datetime import datetime
time_str = datetime.now().strftime('%Y%m%d%H%M%S')
filename = f'profile-{time_str}.png'
drawer = output.GraphvizOutput(output_file=filename)
with PyCallGraph(drawer):
    <code_to_be_profiled>
```

# NumPy

Array manipulation mini language. Can run up to one hundred times faster than equivalent Python code.

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

```
<array> = np.array(<list>)
<array> = np.arange(from_inclusive, to_exclusive, ±step_size)
<array> = np.ones(<shape>)
<array> = np.random.randint(from_inclusive, to_exclusive, <shape>)
```

```
<array>.shape = <shape>
<view>  = <array>.reshape(<shape>)
<view>  = np.broadcast_to(<array>, <shape>)
```

```
<array> = <array>.sum(axis)
indexes = <array>.argmin(axis)
```

- **Shape is a tuple of dimension sizes.**
- **Axis is an index of dimension that gets collapsed. Leftmost dimension has index 0.**

Indexing

```
<el>       = <2d_array>[0, 0]        # First element.
<1d_view>  = <2d_array>[0]           # First row.
<1d_view>  = <2d_array>[:, 0]        # First column. Also [..., 0].
<3d_view>  = <2d_array>[None, :, :]  # Expanded by dimension of size 1.
```

```
<1d_array> = <2d_array>[<1d_row_indexes>, <1d_column_indexes>]
<2d_array> = <2d_array>[<2d_row_indexes>, <2d_column_indexes>]
```

```
<2d_bools> = <2d_array> > 0
<1d_array> = <2d_array>[<2d_bools>]
```

- **If row and column indexes differ in shape, they are combined with broadcasting.**

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

3. If neither non-matching dimension has size 1, rise 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))
<Image> = Image.open('<path>')
<Image> = <Image>.convert('<mode>')
<Image>.save('<path>')
<Image>.show()
```

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

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
size = WIDTH * HEIGHT
hues = [255 * i/size for i in range(size)]
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')
```

ImageDraw

```
from PIL import ImageDraw
```

```
<ImageDraw> = ImageDraw.Draw(<Image>)
<ImageDraw>.point((x, y), fill=None)
<ImageDraw>.line((x1, y1, x2, y2 [, ...]), fill=None, width=0, joint=None)
<ImageDraw>.arc((x1, y1, x2, y2), from_deg, to_deg, fill=None, width=0)
<ImageDraw>.rectangle((x1, y1, x2, y2), fill=None, outline=None, width=0)
<ImageDraw>.polygon((x1, y1, x2, y2 [, ...]), fill=None, outline=None)
<ImageDraw>.ellipse((x1, y1, x2, y2), fill=None, outline=None, width=0)
```

- **Use `'fill=<color>'` to set the primary color.**
- **Use `'outline=<color>'` to set the secondary color.**
- **Colors can be specified as tuple, int, `'#rrggbb'` string or a color name.**

# Animation

Creates a GIF of a bouncing ball:

```python
# $ pip3 install pillow imageio
from PIL import Image, ImageDraw
import imageio
WIDTH, R = 126, 10
frames = []
for velocity in range(15):
    y = sum(range(velocity+1))
    frame = Image.new('L', (WIDTH, WIDTH))
    draw  = ImageDraw.Draw(frame)
    draw.ellipse((WIDTH/2-R, y, WIDTH/2+R, y+2*R), fill='white')
    frames.append(frame)
frames += reversed(frames[1:-1])
imageio.mimsave('test.gif', frames, duration=0.03)
```

# Audio

```python
import wave
```

```python
<Wave_read>  = wave.open('<path>', 'rb')
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.
<bytes>      = <Wave_read>.readframes(nframes)  # Returns next 'nframes' frames.
```

```python
<Wave_write> = wave.open('<path>', 'wb')
<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>.writeframes(<bytes>)               # Appends frames to file.
```

- **Bytes object contains a sequence of frames, each consisting of one or more samples.**
- **In stereo signal first sample of a frame belongs to the left channel.**
- **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, 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
4	-2147483648	0	2147483647

Read Float Samples from WAV File

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

Write Float Samples to WAV File

```python
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(a) for a in float_samples))
```

Examples

Saves a sine wave to a mono WAV file:

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

Adds noise to a mono WAV file:

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

# Synthesizer

Plays Popcorn by Gershon Kingsley:

```python
# $ pip3 install simpleaudio
import simpleaudio, math, struct
from itertools import chain, repeat
F  = 44100
P1 = '71♪,69,,71♪,66,,62♪,66,,59♪,,,'
P2 = '71♪,73,,74♪,73,,74,,71,,73♪,71,,73,,69,,71♪,69,,71,,67,,71♪,,,'
get_pause  = lambda seconds: 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]), 0.25 if '♪' in note else 0.125)
get_frames = lambda note: get_wave(*parse_note(note)) if note else get_pause(0.125)
frames_f   = chain.from_iterable(get_frames(n) for n in f'{P1}{P1}{P2}'.split(','))
frames_b   = b''.join(struct.pack('<h', int(f * 30000)) for f in frames_f)
simpleaudio.play_buffer(frames_b, 1, 2, F)
```

# Basic Script Template

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

from collections import namedtuple
from dataclasses import make_dataclass
from enum import Enum
from sys import argv
import re


def main():
    pass


###
##  UTIL
#

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


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

March 14, 2018 / Jure Šorn