March 14, 2018 / Jure Šorn

Comprehensive Python Cheatsheet

Download text file or Fork me on GitHub.

Main

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

List

```
<list> = <list>[from_inclusive : to_exclusive : step_size]
<list>.append(<el>)
<list>.extend(<list>)
<list> += [<el>]
<list> += <list>
```

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

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

```
index = <list>.index(<el>)  # Returns first index of item.
<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.
<list>.clear()              # Removes all items.
```

Dictionary

```
<view>  = <dict>.keys()
<view>  = <dict>.values()
<view>  = <dict>.items()
<value> = <dict>.get(key, default)         # Returns default if key does not exist.
```

```
<value> = <dict>.setdefault(key, default)  # Same, but also adds default to dict.
<dict>.update(<dict>)
```

```
collections.defaultdict(<type>)     # Creates a dictionary with default value of type.
collections.defaultdict(lambda: 1)  # Creates a dictionary with default value 1.
collections.OrderedDict()           # Creates ordered dictionary.
```

```
dict(<list>)                        # Initiates a dict from list of key-value pairs.
dict(zip(keys, values))             # Initiates a dict from two lists.
{k: v for k, v in <dict>.items() if k in <list>}  # Filters a dict by keys.
```

Counter

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

Set

```
<set> = set()
<set>.add(<el>)
<set>.update(<set>)
<set>.clear()
```

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

Frozenset

Is hashable and can be used as a key in dictionary.

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

Range

```
range(to_exclusive)
range(from_inclusive, to_exclusive)
range(from_inclusive, to_exclusive, step_size)
range(from_inclusive, to_exclusive, -step_size)
```

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

Enumerate

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

Named Tuple

```
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> a = Point(1, y=2)
Point(x=1, y=2)
>>> a.x
1
>>> getattr(a, 'y')
2
>>> Point._fields
('x', 'y')
```

Iterator

Skips first element:

```
next(<iter>)
for element in <iter>:
    ...
```

Reads input until it reaches an empty line:

```
for line in iter(input, ''):
    ...
```

Same, but prints a message every time:

```
from functools import partial
for line in iter(partial(input, 'Please enter value'), ''):
    ...
```

Generator

Convenient way to implement the iterator protocol.

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

```
>>> stepper = step(10, 2)
>>> next(stepper), next(stepper), next(stepper)
(10, 12, 14)
```

Type

```
<type> = type(<el>)  # <class 'int'> / <class 'str'> / ...
```

```
from numbers import Number, Integral, Real, Rational, Complex
is_number   = isinstance(<el>, Number)
is_function = callable(<el>)
```

String

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

```
<list> = <str>.split()                       # Splits on any whitespace character.
<list> = <str>.split(sep=None, maxsplit=-1)  # Splits on 'sep' at most 'maxsplit' times.
<str>  = <str>.join(<list>)                  # Joins elements using string as separator.
```

```
<str>  = <str>.replace(old_str, new_str)
<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>.index(<sub_str>)           # Returns first index of a substring.
<bool> = <str>.isnumeric()                # True if str contains only numeric characters.
<list> = textwrap.wrap(<str>, width)      # Nicely breaks string into lines.
```

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)
```

Print

```
print(<el_1> [, <el_2>, end='', sep='', file=<file>])  # Use 'file=sys.stderr' for errors.
```

```
>>> from pprint import pprint
>>> pprint(locals())
{'__doc__': None,
 '__name__': '__main__',
 '__package__': None, ...}
```

Regex

```
import re
<str>   = re.sub(<regex>, new, text, count=0)  # Substitutes all occurrences.
<list>  = re.findall(<regex>, text)
<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.
<Match_iter> = re.finditer(<regex>, text)      # Searches for all occurrences of pattern.
```

- **Parameter 'flags=re.IGNORECASE' can be used with all functions. Parameter 'flags=re.DOTALL' makes dot also accept newline.**
- **Use '\\1' or r'\1' for backreference.**
- **Use ? to make operators non-greedy.**

Match Object

```
<str> = <Match>.group()   # Whole match.
<str> = <Match>.group(1)  # Part in first bracket.
<int> = <Match>.start()   # Start index of a match.
<int> = <Match>.end()     # Exclusive end index of a match.
```

Special Sequences

Use capital letter 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>)
```

```
>>> Person = namedtuple('Person', 'name height')
>>> person = Person('Jean-Luc', 187)
>>> f'{person.height:10}'
'       187'
>>> '{p.height:10}'.format(p=person)
'       187'
```

General Options

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

Options Specific to Strings

```
{'abcde':.3}    # 'abc'
{'abcde':10.3}  # 'abc       '
```

Options specific to Numbers

```
{1.23456:.3f}    # '1.235'
{1.23456:10.3f}  # '     1.235'
{123456:10,}     # '   123,456'
{123456:10_}     # '   123_456'
{3:08b}          # '00000011' -> Binary with leading zeros.
{3:0<8b}         # '11000000' -> Binary with trailing zeros.
```

Float presentation types:

- `'f'` - Fixed point: `.<precision>f`
- `'e'` - Exponent

Integer presentation types:

- `'c'` - Character
- `'b'` - Binary
- `'x'` - Hex
- `'X'` - HEX

Numbers

Basic Functions

```
round(<num> [, ndigits])
abs(<num>)
```

```python
math.pow(x, y)  # Or: x ** y
```

Constants

```python
from math import e, pi
```

Trigonometry

```python
from math import cos, acos, sin, asin, tan, atan, degrees, radians
```

Logarithm

```python
from math import log, log10, log2
log(x [, base])  # Base e, if not specified.
log10(x)         # Base 10.
log2(x)          # Base 2.
```

Infinity, nan

```python
from math import inf, nan, isfinite, isinf, isnan
```

Or:

```python
float('inf'), float('nan')
```

Random

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

Datetime

```python
from datetime import datetime, strptime
now = datetime.now()
now.month                         # 3
now.strftime('%Y%m%d')            # '20180315'
now.strftime('%Y%m%d%H%M%S')      # '20180315002834'
<datetime> = strptime('2015-05-12 00:39', '%Y-%m-%d %H:%M')
```

Arguments

"*" is the splat operator, that takes a list as input, and expands it into actual positional arguments in the function call.

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

Is the same as:

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

Splat operator can also be used in function declarations:

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

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

And in few other places:

```
>>> a = (1, 2, 3)
>>> [*a]
[1, 2, 3]
```

```
>>> head, *body, tail = [1, 2, 3, 4]
>>> body
[2, 3]
```

Inline

Lambda

```
lambda: <return_value>
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, ..., 9}
<dict> = {i: i*2 for i in range(10)}      # {0: 0, 1: 2, ..., 9: 18}
<iter> = (x+5 for x in range(10))         # (5, 6, ..., 14)
```

```
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, ..., 9)
<any_type> = reduce(lambda sum, x: sum+x, range(10))  # 45
```

Any, All

```
<bool> = any(el[1] for el in <collection>)
```

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, Class

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

from enum import Enum
Direction = Enum('Direction', 'n e s w')
Cutlery = Enum('Cutlery', {'knife': 1, 'fork': 2, 'spoon': 3})

# Warning: Objects will share the objects that are initialized in the dictionary!
Creature = type('Creature', (), {'position': Point(0, 0), 'direction': Direction.n})
creature = Creature()
```

Closure

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

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

Or:

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

Decorator

```
@closure_name
def function_that_gets_passed_to_closure():
    ...
```

Debugger example:

```
from functools import wraps

def debug(func):
    @wraps(func)  # Needed for metadata copying (func name, ...).
    def out(*args, **kwargs):
        print(func.__name__)
        return func(*args, **kwargs)
    return out

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

Class

```
class <name>:
    def __init__(self, a):
        self.a = a
    def __str__(self):
        return str(self.a)
    def __repr__(self):
        return str({'a': self.a})  # Or: return f'{self.__dict__}'

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

Constructor Overloading

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

Copy

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

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()  # Can be used for automatic indexing.
    ...

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

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

```
<member>  = <enum>.<member_name>
<member>  = <enum>['<member_name>']
<member>  = <enum>(<value>)
<name>    = <member>.name
<value>   = <member>.value
```

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

Inline

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

# Functions can not be values, so they must be enclosed in tuple:
LogicOp = Enum('LogicOp', {'AND': (lambda l, r: l and r, ),
                           'OR' : (lambda l, r: l or r, )})

# But 'list(<enum>)' will only work if there is another value in the tuple:
LogicOp = Enum('LogicOp', {'AND': (auto(), lambda l, r: l and r),
                           'OR' : (auto(), lambda l, r: l or r)})
```

System

Arguments

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

Read File

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

Write to File

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

Path

```
import os
<bool> = os.path.exists(<path>)
<bool> = os.path.isfile(<path>)
<bool> = os.path.isdir(<path>)
<list> = os.listdir(<path>)
```

Execute Command

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

Or:

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

Input

```
filename = input('Enter a file name: ')
```

Prints lines until EOF:

```
while True:
    try:
        print(input())
    except EOFError:
        break
```

Recursion Limit

```
>>> sys.getrecursionlimit()
1000
```

```
>>> sys.setrecursionlimit(10000)
```

JSON

```
import json
```

Serialization

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

To preserve order:

```
from collections import OrderedDict
<dict> = json.loads(<str>, object_pairs_hook=OrderedDict)
```

Read File

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

Write to 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)
```

SQLite

```
import sqlite3
db = sqlite3.connect(<filename>)
```

Read

```
cursor = db.execute(<query>)
if cursor:
    cursor.fetchall()  # Or cursor.fetchone()
db.close()
```

Write

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

Pickle

```
import pickle
favorite_color = {'lion': 'yellow', 'kitty': 'red'}
pickle.dump(favorite_color, open('data.p', 'wb'))
favorite_color = pickle.load(open('data.p', 'rb'))
```

Exceptions

```
while True:
    try:
        x = int(input('Please enter a number: '))
    except ValueError:
        print('Oops!  That was no valid number.  Try again...')
    else:
        print('Thank you.')
        break
```

Raising exception:

```
raise ValueError('A very specific message!')
```

Finally:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

Bytes

Bytes objects are immutable sequences of single bytes.

Encode

```
<Bytes> = b'<str>'
<Bytes> = <str>.encode(encoding='utf-8')
<Bytes> = <int>.to_bytes(<length>, byteorder='big|little', signed=False)
<Bytes> = bytes.fromhex(<hex>)
```

Decode

```
<str> = <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):
    with open(filename, 'wb') as file:
        file.write(bytes)
```

```
<Bytes> = b''.join(<list_of_Bytes>)
```

Struct

This module performs conversions between Python values and C struct represented as Python Bytes object.

```
<Bytes> = struct.pack('<format>', <value_1> [, <value_2>, ...])
<tuple> = struct.unpack('<format>', <Bytes>)
```

Example

```
>>> from struct import pack, unpack, calcsize
>>> 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)
>>> calcsize('hhl')
8
```

Format

Use capital leters for unsigned type.

- `'x'` - pad byte
- `'c'` - char
- `'h'` - short
- `'i'` - int
- `'l'` - long
- `'q'` - long long
- `'f'` - float
- `'d'` - double

Hashlib

```
>>> hashlib.md5(<str>.encode()).hexdigest()
'33d0eba106da4d3ebca17fcd3f4c3d77'
```

Threading

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

Thread

```
thread = Thread(target=<function>, args=(<first_arg>, ))
thread.start()
...
thread.join()
```

Lock

```
lock = Rlock()
lock.acquire()
...
lock.release()
```

Itertools

Every function returns an iterator and can accept any collection and/or iterator. If you want to print the iterator, you need to pass it to the list() function.

```
from itertools import *
```

Combinatoric iterators

```
>>> 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')]

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

>>> 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)]
```

Infinite iterators

```
>>> i = count(5, 2)
>>> next(i), next(i), next(i)
(5, 7, 9)

>>> a = cycle('abc')
>>> [next(a) for _ in range(10)]
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c', 'a']

>>> repeat(10, 3)
[10, 10, 10]
```

Iterators

```
>>> chain([1, 2], range(3, 5))
[1, 2, 3, 4]

>>> compress('abc', [True, 0, 1])
['a', 'c']

>>> islice([1, 2, 3], 1, None)  # islice(<seq>, from_inclusive, to_exclusive)
[2, 3]

>>> people = [{'id': 1, 'name': 'bob'},
              {'id': 2, 'name': 'bob'},
              {'id': 3, 'name': 'peter'}]
>>> {name: list(ppp) for name, ppp in groupby(people, key=lambda p: p['name'])}
{'bob':   [{'id': 1, 'name': 'bob'},
           {'id': 2, 'name': 'bob'}],
 'peter': [{'id': 3, 'name': 'peter'}]}
```

Introspection and Metaprograming

Inspecting code at runtime and code that generates code. You can:

- Look at the attributes
- Set new attributes
- Create functions dynamically
- Traverse the parent classes
- Change values in the class

Variables

```
<list> = dir()      # In-scope variables.
<dict> = locals()   # Local variables.
<dict> = globals()  # Global variables.
```

Attributes

```
>>> class Z:
...     def __init__(self):
...         self.a = 'abcde'
...         self.b = 12345
>>> z = Z()
```

```
>>> vars(z)
{'a': 'abcde', 'b': 12345}

>>> getattr(z, 'a')
'abcde'

>>> hasattr(z, 'c')
False

>>> setattr(z, 'c', 10)
```

Parameters

Getting the number of parameters of a function:

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

Type

Type is the root class. If only passed the object it returns it's type. Otherwise it creates a new class (and not the instance!):

```
type(<class_name>, <parents_tuple>, <attributes_dict>)
```

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

MetaClass

Class that creates class:

```
def my_meta_class(name, parents, attrs):
    ...
    return type(name, parents, attrs)
```

Or:

```
class MyMetaClass(type):
    def __new__(klass, name, parents, attrs):
        ...
        return type.__new__(klass, name, parents, attrs)
```

Metaclass Attribute

When class is created it checks if it has metaclass defined. If not, it recursively checks if any of his parents has it defined, and eventually comes to type:

```
class BlaBla:
    __metaclass__ = Bla
```

Operator

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

```
from enum import Enum
from functools import reduce

product_of_elems = reduce(mul, <list>)
sorted_by_second = sorted(<list>, key=itemgetter(1))
sorted_by_both   = sorted(<list>, key=itemgetter(0, 1))
LogicOp          = Enum('LogicOp', {'AND': (and_, ),
                                    'OR' : (or_, )})
```

Eval

Basic

```
>>> from ast import literal_eval
>>> literal_eval('1 + 1')
2
>>> literal_eval('[1, 2, 3]')
[1, 2, 3]
```

Detailed

```
from ast import parse, Num, BinOp, UnaryOp, \
                Add, Sub, Mult, Div, Pow, BitXor, USub
import operator as op

operators = {Add:    op.add,
             Sub:    op.sub,
             Mult:   op.mul,
             Div:    op.truediv,
             Pow:    op.pow,
             BitXor: op.xor,
             USub:   op.neg}

def evaluate(expression):
    root = parse(expression, mode='eval')
    return eval_node(root.body)

def eval_node(node):
    type_ = type(node)
    if type_ == Num:
        return node.n
    if type_ not in [BinOp, UnaryOp]:
        raise TypeError(node)
    operator = operators[type(node.op)]
    if type_ == BinOp:
        left, right = eval_node(node.left), eval_node(node.right)
        return operator(left, right)
    elif type_ == UnaryOp:
        operand = eval_node(node.operand)
        return operator(operand)
```

```
>>> evaluate('2^6')
4
>>> evaluate('2**6')
64
>>> evaluate('1 + 2*3**(4^5) / (6 + -7)')
-5.0
```

Coroutine

- Similar to Generator, but Generator pulls data through the pipe with iteration, while Coroutine pushes data into the pipeline with send().
- Coroutines provide more powerful data routing possibilities than iterators.
- If you built a collection of simple data processing components, you can glue them together into complex arrangements of pipes, branches, merging, etc.

Helper Decorator

- All coroutines must be "primed" by first calling .next()
- Remembering to call .next() is easy to forget.
- Solved by wrapping coroutines with a decorator:

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

Pipeline Example

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

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

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

reader(adder(printer()))
```

Libraries

Plot

```
# $ pip3 install matplotlib
from matplotlib import pyplot
pyplot.plot(<data_1> [, <data_2>, ...])
pyplot.show()
pyplot.savefig(<filename>, transparent=True)
```

Table

Prints CSV file as ASCII table:

```
# $ pip3 install tabulate
import csv
from tabulate import tabulate
with open(<filename>, newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=';')
    headers = [a.title() for a in next(reader)]
    print(tabulate(reader, headers))
```

Curses

```
# $ pip3 install curses
from curses import wrapper

def main():
    wrapper(draw)

def draw(screen):
    screen.clear()
    screen.addstr(0, 0, 'Press ESC to quit.')
    while screen.getch() != 27:
        pass

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

Image

Creates PNG image of greyscale gradient:

```
# $ pip3 install pillow
from PIL import Image
width, height = 100, 100
img = Image.new('L', (width, height), 'white')
img.putdata([255*a/(width*height) for a in range(width*height)])
img.save('out.png')
```

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.

Audio

Saves list of floats with values between 0 and 1 to a WAV file:

```
import wave, struct
frames = [struct.pack('h', int((a-0.5)*60000)) for a in <list>]
wf = wave.open(<filename>, 'wb')
wf.setnchannels(1)
wf.setsampwidth(4)
wf.setframerate(44100)
wf.writeframes(b''.join(frames))
wf.close()
```

Url

```
from urllib.parse import quote, quote_plus, unquote, unquote_plus
```

Encode

```
>>> quote("Can't be in URL!")
'Can%27t%20be%20in%20URL%21'
>>> quote_plus("Can't be in URL!")
'Can%27t+be+in+URL%21'
```

Decode

```
>>> unquote('Can%27t+be+in+URL%21')
"Can't+be+in+URL!"'
>>> unquote_plus('Can%27t+be+in+URL%21')
"Can't be in URL!"
```

Web

```
# $ pip3 install bottle
import bottle
from urllib.parse import unquote
```

Run

```
bottle.run(host='localhost', port=8080)
bottle.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):
    sport = unquote(sport).lower()
    page = read_file(sport)
    return template(page)
```

REST request

```python
@post('/odds/<sport>')
def odds_handler(sport):
    team = bottle.request.forms.get('team')
    team = unquote(team).lower()

    db = sqlite3.connect(<db_path>)
    home_odds, away_odds = get_odds(db, sport, team)
    db.close()

    response.headers['Content-Type'] = 'application/json'
    response.headers['Cache-Control'] = 'no-cache'
    return json.dumps([home_odds, away_odds])
```

Profile

Basic:

```python
from time import time
start_time = time()
...
duration = time() - start_time
```

Times execution of the passed code:

```python
from timeit import timeit
timeit('"-".join(str(n) for n in range(100))',
       number=10000, globals=globals())
```

Generates a PNG image of call graph and highlights the bottlenecks:

```python
# $ pip3 install pycallgraph
import pycallgraph
graph = pycallgraph.output.GraphvizOutput()
graph.output_file = get_filename()
with pycallgraph.PyCallGraph(output=graph):
    <code_to_be_profiled>
```

```python
def get_filename():
    from datetime import datetime
    time_str = datetime.now().strftime('%Y%m%d%H%M%S')
    return f'profile-{time_str}.png'
```

Progress Bar

Tqdm

```python
# $ pip3 install tqdm
from tqdm import tqdm
from time import sleep
for i in tqdm(range(100)):
    sleep(0.02)
for i in tqdm([1, 2, 3]):
    sleep(0.2)
```

Basic

```python
import sys
```

```python
class Bar():
    @staticmethod
    def range(*args):
        bar = Bar(len(list(range(*args))))
        for i in range(*args):
            yield i
            bar.tick()
    @staticmethod
    def foreach(elements):
        bar = Bar(len(elements))
        for el in elements:
            yield el
            bar.tick()
    def __init__(s, steps, width=40):
        s.st, s.wi, s.fl, s.i = steps, width, 0, 0
        s.th = s.fl * s.st / s.wi
        s.p(f"[{' ' * s.wi}]")
        s.p('\b' * (s.wi + 1))
    def tick(s):
        s.i += 1
        while s.i > s.th:
            s.fl += 1
            s.th = s.fl * s.st / s.wi
            s.p('-')
        if s.i == s.st:
            s.p('\n')
    def p(s, t):
        sys.stdout.write(t)
        sys.stdout.flush()
```

Usage:

```python
from time import sleep
for i in Bar.range(100):
    sleep(0.02)
for el in Bar.foreach([1, 2, 3]):
    sleep(0.2)
```

Basic Script Template

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

from collections import namedtuple
from enum import Enum
import re
import sys


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