

Python3 Quick Reference

Notes on advanced features in Python 3.7 covering functional programming, external data structures, and tricks on OOP.

Ref: [Python3 Cookbook](#), [Python Standard Library](#).

Shu Wang, June 2019

Python3 Quick Reference

Data Structure & FP

- heapq
- collections
- itertools/iterator
- coroutine/asyncio
- functions
- decorator
- misc

Object-Oriented Programming

- magics
- property
- descriptor
- misc

Acceleration

Data Science

- dplyr vs pandas

Data Structure & FP

heapq

```
# basic usage
>>> import heapq
>>> nums = [3, 4, 1, 2]
>>> heapq.heapify(nums) # inplace
>>> heapq.heappop(nums) # smallest first pop out
# must be a heap, will heapify automatically, O(log N)
>>> heapq.nlargest(3, nums) # good when n < N
# if n = 1, use min/max, if n ~ N, sorted(items)[:n]
>>> heapq.merge(list_a, list_b) # return iterator, can do very LOOOOG SORTED series
>>> portfolio = [{'name': 'IBM', 'shares': 100, 'price': 91.1},
                 {'name': 'AAPL', 'shares': 50, 'price': 543.22},...]
>>> cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
# same structure: sorted(list, key=typing.Callable)

# use heapq to build a priority queue # Ch1.5
>>> class PriorityQueue:
    def __init__(self):
        self._queue = [] # private, list of tuples
        self._index = 0 # keep order with same priority
    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        # compare based on the order in the tuple
        self._index += 1 # save the insert order internally
    def pop(self): return heapq.heappop(self._queue)[-1] # last one in tuple
>>> class Item:
    def __init__(self, name): self.name = name
    def __repr__(self): return 'Item({!r})'.format(self.name)
>>> q = PriorityQueue(); q.push(Item('foo'), 1); q.push(Item('bar'), 5); q.pop()
Item('bar')
# https://docs.python.org/3/library/heapq.html

# heap sort quick note
>>> def sort(arr):
    def scan_down(start, end, arr):
        # heapify when the first element is wrong
        root = start
        while True:
            child = 2 * root + 1
            if child > end: break # out of range
            if child + 1 <= end and arr[child] < arr[child + 1]:
                child += 1 # pick the larger child
            if arr[root] < arr[child]: # maintain this partial area
                arr[root], arr[child] = arr[child], arr[root]
                root = child # move down, leave the upper level DONE
            else: break
        for start in range((len(arr) - 2) // 2, -1, -1):
            # from the lowest son to root going up
            # make sure when scan_down, only care about the current area
            scan_down(start, len(arr) - 1, arr)
            # in each node look down to the lowest level
            # time complexity O(NlogN)
        for end in range(len(arr) - 1, 0, -1):
            arr[0], arr[end] = arr[end], arr[0] # finish one
            scan_down(0, end - 1, arr) # heapify
        return arr
```

collections

```
# collections.deque(maxlen=N) # Ch1.3
# append()/appendleft()/pop()/popleft()
# file pattern-matching example
>>> from collections import deque
>>> def search(lines, pattern, history=5):
    # generator for mapping patterns
    previous_lines = deque(maxlen=history)
    # hidden variable storing key info
    for line in lines:
        if pattern in line:
            yield line, previous_lines
            previous_lines.append(line)
>>> with open(r'../..../cookbook/somefile.txt') as f:
    for line, prevlines in search(f, 'python', 5):
        for pline in prevlines: print(pline, end='')
        print(line, end='')

# collections.defaultdict(list/set) # Ch1.6
>>> from collections import defaultdict
>>> d = defaultdict(list/set)
```

```
>>> d['a'].append(1); d['a'].append(2); d['a']; ...;
# alternative: d = {}; d.setdefault('a', []).append(1)
# inplace, now d = {'a': [1]}

# collections.OrderedDict # Ch1.7 # keep the insert order
# memory cost, twice as large as a normal dict, one additional list

# collections.Counter(iterable) # Ch1.12 # return a dict
# support math opt, +/-

# collections.namedtuple # Ch1.18
# simplify the code or even replace the use of dict
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub.addr # 'jonesy@example.com'
>>> sub._replace(addr='shuw@mit.edu') # _replace(**dict)
# immutable, only change through _replace()

# collections.ChainMap # Ch1.20
# combine multiple iterables, logically a dict but not really merged
# only an intermediate object connecting dicts, support len, keys(), values()
# if multiple values, return the first one always
>>> from collections import ChainMap
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> c = ChainMap(a,b)
>>> print(c['x']) # Outputs 1 (from a)

# https://docs.python.org/3/library/collections.html
```

itertools/iterator

```
# itertools.groupby # Ch1.15
>>> from operator import itemgetter
>>> from itertools import groupby
>>> rows = [{'address': ..., 'date': ...}, ...] # list of dict
>>> rows.sort(key=itemgetter('date')) # sort by date
>>> for date, items in groupby(rows, key=itemgetter('date')):
    for i in items: print(' ', i) # items: itertools._grouper, iterable
    # if groupby multiple: itemgetter('x', 'y'), then date is tuple
    # if only groupby(rows), then date is rows.keys() by default, dict
    # if don't care about memory mgmt, can create a separate data structure to store

# itertools.compress(iterable, bool_selector) # Ch1.16
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> list(compress(addresses, more5)) # return an iterator
# same return type as map/filter/reduce

# iterator
# use next() till StopIteration, next(f, None) returns None when done
# customized object, want to do iteration on it directly: __iter__(), iter(iterable)
# iterator: __iter__() defined return an object with __next__() defined
# iterable: __iter__() defined return an iterator, each time calling __iter__ having a new iterator
# for: work for both iterator and iterable, call __iter__()
# iter(s) <=> s.__iter__() # a generator
# reverse(s) <=> s.__reversed__() # a generator

# itertools.islice # Ch4.7 # note that generator is irreversible
>>> from itertools import islice
>>> def count(n): while True: yield n; n += 1
>>> c = count(0) # cannot do c[10:20]
>>> [print x for x in islice(c, 10, 20)]

# itertools.dropwhile(callable, iterator) # drop til first False # Ch4.8
# itertools.permutations/combinations(iterable, n) # Ch4.9
# itertools.chain(a, b, c, ...) returns an iterator # Ch4.12

# iter() to have while # Ch4.16
>>> import sys
>>> f = open('/etc/passwd')
>>> for chunk in iter(lambda: f.read(10), ''): n = sys.stdout.write(chunk)
# keep calling callable till the returned value equals to the stopping value param

# misc in for
# for x in chain(a, b) more efficient than for x in a + b (create a new arr)
# enumerate(iterable, start_index)
# for x, y in zip(xs, ys): ... # iterate multiple series in sync
# dict/list(zip(xs, ys)) to transform the iterator to dict/list

# itertools.accumulate(see functions)/chain/dropwhile/filterfalse/starmap/takewhile/zip_longest
# itertools.product/permutations/combinations
# https://docs.python.org/3/library/itertools.html
# http://book.pythontips.com/en/latest/map_filter.html
```

coroutine/asyncio

```
# generator: more flexible way of building iteration tools
>>> def frange(start, stop, increment):
    x = start # store hidden variables
    while x < stop: # control flow
        yield x # each time __next__() stops here
        x += increment # start from here for next __next__()

# iteration for an object using generator, iterator protocol # Ch4.4
# for loop as a consumer of the data flow, generator as the producer # Ch4.13
# DFS/BFS for a tree using generator
>>> import collections
>>> class Node:
    def __init__(self, value):
        self._value = value # true embedded value
        self._children = [] # reference
    def __repr__(self): return 'Node({!r})'.format(self._value)
    def __iter__(self): return iter(self._children) # proxy iterator
    def add_child(self, node): self._children.append(node) # add a ref
    def depth_first(self):
        yield self # check the current node first
        for c in self: # for each in children, go in and iterate
            yield from c.depth_first() # a generator is also an iterator
    def breadth_first(self, bfs_children=collections.deque()):
        # BFS using a deque: append() the first node in;
        # while T: popleft() a node, visit, append() it's children, till empty
        # behavior of the root is different from the children, ext info required (param)
        # NOTE: pending issue: step in for loop in an empty set
        # NOTE: using generator can save a huge amount of code
        yield self # first check the current node
        [bfs_children.append(c) for c in self] # fill the deque
        deque_len = len(bfs_children) # update the current len AFTER append
        while deque_len > 0: # when empty no more yield, generator stop
            # can't call deque directly: RuntimeError: deque mutated during iteration
            node = bfs_children.popleft() # pop a new node in iteration
            yield from node.breadth_first(bfs_children) # pull yields from sub-generator
            deque_len = len(bfs_children) # len might be higher than above
            # print will executed after done instead of in each step

>>> root = Node(2); child1 = Node(3); child2 = Node(5)
>>> root.add_child(child1); root.add_child(child2)
>>> child1.add_child(Node(11)); child1.add_child(Node(13)); child2.add_child(Node(23))
>>> for ch in root.depth_first(): print(ch) # 2, 3, 11, 13, 5, 23
>>> for ch in root.breadth_first(): print(ch) # 2, 3, 5, 11, 13, 23
# TODO: how about for graphs? additional node 'visited' status and drop_duplicates
# https://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/

# flatten a nested series
>>> from collections import Iterable
>>> def flatten(items, ignore_types=(str, bytes)):
    for x in items: # outer layer iterable with __iter__() defined
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            # if str and bytes don't keep flattening
            yield from flatten(x) # pull up the yields in the next level
        else: yield x

# inline callbacks & coroutine # Ch7.11/Ch12.12 (HARD, VERY CONCISE AND ELEGANT)
# adv of coroutine: running multiple function (process) in parallel in programming level
# an even more advanced feature new in 3.7: @asyncio.coroutine + yield from / async + await
# https://juejin.im/post/5c13245ee51d455fa5451f33
# http://www.dabeaz.com/coroutines/index.html
# https://www.geeksforgeeks.org/coroutine-in-python/
# " concerned about the proliferation of small functions and mind boggling control flow.
# want some way to make the code look more like a normal sequence of procedural steps. "
>>> from queue import Queue
>>> from functools import wraps
>>> def apply_async(func, args, *, callback): # key part with callback
    result = func(*args) # do sth with input params
    callback(result) # apply callback on it
>>> class Async: # pack up the func and args
    def __init__(self, func, args):
        self.func = func; self.args = args
>>> def inlined_async(func): # decorator for the WHOLE CONTROL FLOW
    @wraps(func) # transfer function attributes
    def wrapper(*args):
        f = func(*args) # coroutine: test() defined below, lazy computation, nothing happens now
        result_queue = Queue(); result_queue.put(None)
        # push, first send has to be a None, g.send(None) <=> g.next()
        while True:
            result = result_queue.get() # pop
            try:
                a = f.send(result) # get the struct from test()
                # DATA EXCHANGE! communication between external code and coroutine
```

```

        # result become the value of yield expression
        # raise StopIteration when finished
        apply_async(a.func, a.args, callback=result_queue.put) # !!!
        # callback(put) will be executed after the yield, and the
        # result will be sent in next iteration, that's why using QUEUE here
        # a.func(*a.args) embedded, callback manipulating same object
    except StopIteration: break
    return wrapper
>>> @inlined_async
def test(): # core control flow with clearer structure, coroutine
    def add(x, y): return x + y
    # yield expression! turn a generator to a rudimentary coroutine
    # produce a value and stop UNTIL __next__() or send() IS CALLED
    # when send() is called, r take the value from outside
    # hide complicated control flow behind the generator
    r = yield Async(add, (2, 3)); print(r) # 5
    # ! this r takes the value from the second .send() instead of the first
    # the first .send() only execute the code till yield, so must be .send(None)!
    r = yield Async(add, ('hello', 'world')); print(r) # helloworld
    for n in range(10): r = yield Async(add, (n, n)); print(r) # 0 \n 2 ... 18

# asyncio: putting coroutines into an event-loop
>>> import threading
>>> import asyncio
>>> @asyncio.coroutine
def hello():
    # multiple threading for coroutines
    print('Hello world! (%s)' % threading.currentThread())
    yield from asyncio.sleep(1) # won't wait for asyncio.sleep(1)
    print('Hello again! (%s)' % threading.currentThread())
>>> loop = asyncio.get_event_loop()
>>> tasks = [hello(), hello()] # sync in one thread
>>> loop.run_until_complete(asyncio.wait(tasks)) # right here we wait
>>> loop.close()
# outputs: print; print; (first prints in two) (stop for about 1 sec) print; print (second prints)
# PENDING: RuntimeError: this event loop is already running

```

functions

```

# variable params # Ch7.2
# def a(x, *args, y[, **kwargs]): pass / def a(x, *, y[, **kwargs])
# y can be after *args, forced to be kw, have better structure than **kwargs

# meta info, f.__annotations__ written # Ch7.3
# def f(x:typing.XXX=XXX, y:typing.XXX=XXX) -> typing.XXX: pass

# misc on default values for params # Ch7.5
# default value only assigned once for a function
# default value should be immutable, don't do []
# use if a is None, instead of if not a
# use a new class(object) for identity check

# put lambda in a list comp # Ch7.7
>>> funcs = [lambda x: x+n for n in range(5)]
>>> for f in funcs: print(f(0)) # only record the last n
>>> funcs = [lambda x, n=n: x+n for n in range(5)] # alt
>>> for f in funcs: print(f(0)) # embed the value in param list

# for-loop killer: partial + list(map(f, iterable)) / list(accumulate([snap] + feeds, f))
# alt: use generator to manage namespace and iteration flow, get better memory mgmt
# itertools.accumulate is ROUGHLY equiv to the following generator:
>>> def accumulate(iterable, func=operator.add):
    it = iter(iterable) # iterable to iterator
    try: total = next(it) # check iterable empty first
    except StopIteration: return
    yield total # the first element
    for element in it: total = func(total, element); yield total
    # also we have a for, but clearer and more concise

# enclosure for storing additional info for a function # Ch7.9
# provide an independent environment
>>> class UrlTemplate: # class enclosure
    def __init__(self, template): self.template = template
    def open(self, **kwargs): return urlopen(self.template.format_map(kwargs))
>>> def urltemplate(template): # function enclosure, more elegant!
    # define anything should be wrapped in this func's environment
    def opener(**kwargs): return urlopen(template.format_map(kwargs))
    return opener

# multiple calls on callback, add info in callback to use # Ch7.10 (HARD)
# option1: create a new class, use method as callback
# option2: def f(): var=0; def g(): do_sth(); return g
# option3: coroutine:
>>> def make_handler():

```

```

sequence = 0
while True:
    result = yield
    sequence += 1
    print('[{}] Got: {}'.format(sequence, result))
>>> handler = make_handler()
>>> next(handler) # Advance to the yield, MUST DO before using send()
>>> apply_async(add, (2, 3), callback=handler.send) # use send() as callback
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler.send)
[2] Got: helloworld

# extra operations on embedded data in enclosure # Ch7.12
>>> def sample(): # very similar to cls but simpler
    n = 0 # hidden var
    def func(): print('n=', n) # closure func
    def get_n(): return n
    def set_n(value): # jump out
        nonlocal n
        n = value
    func.get_n, func.set_n = get_n, set_n # func attr!
    return func

```

decorator

```

# keep the meta info/attributes when using decorator # Ch9.2
>>> import time
>>> from functools import wraps
>>> def timethis(func):
    @wraps(func) # __name__/doc/annotations__
    # can use __wrapped__ to get info from the core func
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        return result
    return wrapper

# decorator with params, double wrapping # Ch9.4
>>> from functools import wraps
>>> import logging
>>> def logged(level, name=None, message=None):
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
        return wrapper
    return decorate
>>> @logged(logging.CRITICAL, 'example')
def spam(): print('Spam!')

# alt example for double wrapping in TGO
>>> def null_wrapper(check_param_args:int=0) -> typing.Callable:
    def decorator(func: typing.Callable) -> typing.Callable:
        def wrapper(*args, **kwargs):
            if len(args[check_param_args]) == 0: return {}
            # check the input, enter empty dict thread if input is empty
            else:
                try: return func(*args, **kwargs)
                except Exception as e:
                    print('Error:', e)
                    return {}
        return wrapper
    return decorator
>>> @null_wrapper(0)
def f(): pass # same as f = null_wrapper(0)(f)

# more on decorators, pending, Ch9.5+

```

misc

```

# unpacking and assigning
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, (year, mon, day) = data
>>> _, shares, price, _ = data
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_, (*_, year) = record # * will create a list
>>> s = 'Hello'; a, b, c, d, e = s

```

```

# math op only on keys, need to flip if on values, or apply callable # Ch1.8
>>> prices = {'ACME': 45.23, 'AAPL': 612.78, 'IBM': 205.55, 'FB': 10.75}
>>> min(zip(prices.values(), prices.keys())) # a tuple
>>> min(prices, key=lambda k: prices[k]) # only the key
>>> sorted(zip(prices.values(), prices.keys()))

# keys()/items() support set op, &/-, values() might have dups, to set first
# use named slice to avoid hard-coding indexing
>>> a = slice(5, 50, 2)
>>> s = 'Helloworld'; a.indices(len(s)) # to truncate a and return a tuple
>>> s[slice(*a.indices(len(s)))] # use the tuple to build slice to index # ugly

# sorted/min/max(list, callable)!
>>> from operator import itemgetter, attrgetter # attrgetter for cls
# work with any object with __getitem__(), multiple inputs yield a tuple
>>> rows_by_fname = sorted(rows, key=itemgetter('fname', 'lname'))
>>> rows_by_fname = sorted(rows, key=lambda r: (r['fname'], r['lname'])) # slower
# sort by one value of each element in the compounded list

# build subset based on values
>>> prices = {'tkr': price}
>>> p1 = {key: value for key, value in prices.items() if value > 200} # 2x faster
>>> p1 = dict((key, value) for key, value in prices.items() if value > 200)

# aggregate after transformation
>>> s = sum([x * x for x in nums]) # memory ineff
>>> s = sum((x * x for x in nums)) # generator basic
>>> s = sum(x * x for x in nums) # equiv, elegant!

```

Object-Oriented Programming

magics

```
# __str__() for print()/user, __repr__() for direct print/dev: eval(repr(x)) == x is True # Ch8.1
# __format__() string formatting # Ch8.2
# __slots__() for memory mgmt # Ch8.4
# super() for father class methods, used in __get/setattr__ # Ch8.7, more details on cls.__mro__

# context manager # Ch8.3
# __enter__(), __exit__() for with
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> from functools import partial
>>> class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = family
        self.type = type
        self.sock = None
    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock
    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
>>> conn = LazyConnection(('www.python.org', 80))
# Connection closed
>>> with conn as s:
    # conn.__enter__() executes: connection open
    s.send(b'GET /index.html HTTP/1.0\r\n')
    s.send(b'Host: www.python.org\r\n')
    s.send(b'\r\n')
    resp = b''.join(iter(partial(s.recv, 8192), b''))
    # conn.__exit__() executes: connection closed
    # will execute till the end no matter what happen
    # exceptions can be included in __exit__

# alternative context manager # Ch9.22
>>> import time
>>> from contextlib import contextmanager
>>> @contextmanager # ONLY for self-included function, no external ops
    def timethis(label):
        start = time.time() # __enter__()
        try: yield # before this wrapped as __enter__()
        finally: # after this wrapped as __exit__()
            end = time.time()
            print('{}: {}'.format(label, end - start))
# example use
>>> with timethis('counting'):
    n = 10000000
    while n > 0:
        n -= 1
# a more advanced example
>>> @contextmanager
    def list_transaction(orig_list):
        working = list(orig_list)
        yield working
        orig_list[:] = working
>>> items = [1, 2, 3]
>>> with list_transaction(items) as working:
    working.append(4)
    working.append(5)
>>> items # [1, 2, 3, 4, 5]
# only valid when no exception
```

property

```
# additional checks when setting/getting
>>> class Person:
    def __init__(self, name): self._name = name
    @property
    def name(self): return self._name
    @name.setter
    def name(self): sth() self._name = name
    @name.deleter
    def name(self): raise AttributeError("Can't delete attribute")
# link existing functions: name = property(get_name, set_name, del_name) # use name to access
# do not write @property without additional operations
```



```
# in son cls, either rewrite @property @xxx.getter @xxx.setter,
# or hard-code the name of the class @person.name.setter to rewrite
# if don't know the name of base, only way is to rewrite all @property and use super()
```

descriptor

```
# a new simplistic class with basic op overloaded
# __get/set/delete__ used for @classmethod @staticmethod @property __slots__
class Integer:
    def __init__(self, name):
        self.name = name
    def __get__(self, instance, cls):
        if instance is None: return self
        else: return instance.__dict__[self.name]
    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        instance.__dict__[self.name] = value
    def __delete__(self, instance):
        del instance.__dict__[self.name]

# a more complicated case
# Descriptor for a type-checked attribute
class Typed:
    def __init__(self, name, expected_type):
        self.name = name
        self.expected_type = expected_type
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]
    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('Expected ' + str(self.expected_type))
        instance.__dict__[self.name] = value
    def __delete__(self, instance):
        del instance.__dict__[self.name]
# class decorator that applies it to selected attributes
def typeassert(**kwargs):
    def decorate(cls):
        for name, expected_type in kwargs.items():
            # Attach a Typed descriptor to the class
            setattr(cls, name, Typed(name, expected_type))
        return cls
    return decorate
@typeassert(name=str, shares=int, price=float)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

# a descriptor with overloading @property
class String:
    def __init__(self, name):
        self.name = name
    def __get__(self, instance, cls):
        if instance is None:
            return self
        return instance.__dict__[self.name]
    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        instance.__dict__[self.name] = value
class Person:
    name = String('name') # carrying a descriptor
    def __init__(self, name):
        self.name = name
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name
    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)
    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)
```

misc

```
__sizeof__(), __len__(), __format__()  
__getattr__(self, name), __setattr__(self, name, value), __delattr__(self, name)
```

Acceleration

```
# must first understand the bottleneck of running speed before speeding up the code
# write simple decorator to get info about speed # Ch14.13
# Unix: time python3 test.py / python3 -m cProfile test.py / timeit in python

# tips to speed it up!
# don't optimize
# only optimize the bottleneck and put robustness/readability in the first place
# algorithm goes first, improve time complexity first
# use functions to create local environments
# get rid of attribute calls as much as possible (__getattr__ very expensive)
# use a local var to catch global var that's frequently called
# avoid unnecessary wrappers
# use C-based build-in vessels
```

Data Science

dplyr vs pandas

dplyr	pandas
Querying, Filtering, Sampling	
<code>dim(df)</code>	<code>df.shape</code>
<code>head(df)</code>	<code>df.head()</code>
<code>slice(df, 1:10)</code>	<code>df.iloc[:9]</code>
<code>filter(df, col1 == 1, col2 == 1)</code>	<code>df.query('col1 == 1 & col2 == 1')</code>
<code>df[df\$col1 == 1 & df\$col2 == 1,]</code>	<code>df[(df.col1 == 1) & (df.col2 == 1)]</code>
<code>select(df, col1, col2)</code>	<code>df[['col1', 'col2']]</code>
<code>select(df, col1:col3)</code>	<code>df.loc[:, 'col1':'col3']</code>
<code>select(df, -(col1:col3))</code>	<code>df.drop(cols_to_drop, axis=1)</code>
<code>distinct(select(df, col1))</code>	<code>df[['col1']].drop_duplicates()</code>
<code>distinct(select(df, col1, col2))</code>	<code>df[['col1', 'col2']].drop_duplicates()</code>
<code>sample_n(df, 10)</code>	<code>df.sample(n=10)</code>
<code>sample_frac(df, 0.01)</code>	<code>df.sample(frac=0.01)</code>
Sorting	
<code>arrange(df, col1, col2)</code>	<code>df.sort_values(['col1', 'col2'])</code>
<code>arrange(df, desc(col1))</code>	<code>df.sort_values('col1', ascending=False)</code>
Transforming	
<code>select(df, col_one = col1)</code>	<code>df.rename(columns={'col1': 'col_one'})['col_one']</code>
<code>rename(df, col_one = col1)</code>	<code>df.rename(columns={'col1': 'col_one'})</code>
<code>mutate(df, c=a-b)</code>	<code>df.assign(c=df.a-df.b)</code>
Grouping and Summarizing	
<code>summary(df)</code>	<code>df.describe()</code>
<code>gdf <- group_by(df, col1)</code>	<code>gdf = df.groupby('col1')</code>
<code>summarise(gdf, avg=mean(col1, na.rm=TRUE))</code>	<code>df.groupby('col1').agg({'col1': 'mean'})</code>
<code>summarise(gdf, total=sum(col1))</code>	<code>df.groupby('col1').sum()</code>