In [1]:

```
from io import StringIO
import sys
class Capturing(list):
    def __enter__(self):
        self._stdout = sys.stdout
        sys.stdout = self._stringio = StringIO()
        return self

    def __exit__(self, *args):
        self.extend(self._stringio.getvalue().splitlines())
        del self._stringio    # free up some memory
        sys.stdout = self._stdout
```

Question 1

In [2]:

```
def profile_nt():
    '''
    This function generates profiles for 100 individuals. The profiles are stored
    in named tuples. The most common blood type, oldest age, mean age, and the
    mean location are printed.
    '''
    ########################################
    ##############  Imports  ###############
    ########################################
    from faker import Faker
    from collections import namedtuple
    from collections import Counter
    import datetime
    from time import perf_counter

    ########################################
    ###### Intitializing the Objects ########
    ########################################
    fake = Faker()
    blood_count = Counter()


    ########################################
    ####### Defining the NamedTuple #########
    ########################################
    profile = namedtuple('profile', ['job', 'company', 'ssn', 'residence', 'current
                         'blood_group', 'website', 'username', 'name', 'sex', 'addre
    profile.__doc__ = 'Fake personnel profile using faker library'

    start = perf_counter()
    ########################################
    ######## Creating the Profiles ##########
    ########################################
    cnt = 10000
    for c in range(cnt):
        globals()['profile' + str(c)] = profile(**fake.profile())


    lat = 0
    lng = 0
    mindob = datetime.date(datetime.datetime.now().year,
                           datetime.datetime.now().month, datetime.datetime.now().day)
    sumdob = datetime.timedelta(0)
    today = datetime.date(datetime.datetime.now().year,
                          datetime.datetime.now().month, datetime.datetime.now().day)
    for c in range(cnt):
        blood_count.update([(globals()['profile' + str(c)]).blood_group])
        lat += (globals()['profile' + str(c)]).current_location[0]
        lng += (globals()['profile' + str(c)]).current_location[1]
        mindob = min(mindob, (globals()['profile' + str(c)]).birthdate)
        sumdob += today - (globals()['profile' + str(c)]).birthdate

    lat = lat/cnt
    lng = lng/cnt

    sumdob = int(sumdob.days / cnt)
    avg_year = sumdob // 365
    avg_mnt = (sumdob - avg_year * 365) // 30
    avg_day = (sumdob - avg_year * 365 - avg_mnt*30)
```

```
    max_age = (today - mindob).days
    max_year = max_age // 365
    max_mnt = (max_age - max_year * 365) // 30
    max_day = round((max_age - max_year * 365 - max_mnt*30),0)

    common_bt = blood_count.most_common(1)[0][0]
    common_bt_cnt = blood_count.most_common(1)[0][1]

    print(
        f'The most common blood type is {common_bt} with {common_bt_cnt} counts')
    print(f'Avg. Age is {avg_year} Years, {avg_mnt} Months, and {avg_day} Days')
    print(f'Oldest. Age is {max_year} Years, {max_mnt} Months, and {max_day} Days')
    print(f'Mean location is ({lat}, {lng})')
    end = perf_counter()

    total_elapsed = end - start
    print(f'Time {total_elapsed}')

    return {'count': cnt, 'common_bt_cnt': common_bt_cnt, 'avg_year': avg_year, 'av
```

In [3]:

```
a = profile_nt()
```

```
The most common blood type is A+ with 1281 counts
Avg. Age is 57 Years, 5 Months, and 13 Days
Oldest. Age is 116 Years, 0 Months, and 21 Days
Mean location is (-0.18607559955, -0.0880786109)
Time 11.361591549999503
```

Test Cases

In [4]:

```
def test_profile_nt():
    a = profile_nt()
    assert a['count'] == 10000, "Number of profiles must be 10000!"
    assert a['common_bt_cnt'] <= a['count'], "Max blood type count exceeds number o
    assert [True if a['max_year'] > a['avg_year'] else (True if a['max_mnt'] > a['a
        True if a['max_day'] > a['avg_day'] else False))], "How can the max age be
    assert len(a.keys()) >= 8, 'Insufficient number of outputs in your code!'
    assert sum(number < 0 for number in [a['max_year'], a['max_mnt'], a['max_day'],
                                         a['avg_year'], a['avg_mnt'], a['avg_day']]
    assert profile_nt.__doc__, "Your function must have a docstring"

    with Capturing() as output:
        profile_nt()
    assert any(["Time" in o for o in output]), "How will you compare the time with
```

In [5]:

```
test_profile_nt()
```

```
The most common blood type is B+ with 1312 counts
Avg. Age is 57 Years, 9 Months, and 7 Days
Oldest. Age is 116 Years, 0 Months, and 16 Days
Mean location is (0.16693314585, 0.7561320293)
Time 12.710850941999524
```

Question 2

In [6]:

```
def profile_dict():
    '''
    This function generates profiles for 100 individuals. The profiles are stored
    in dictionaries. The most common blood type, oldest age, mean age, and the
    mean location are printed.
    '''
    ########################################
    ##############  Imports  ###############
    ########################################
    from faker import Faker
    from collections import Counter
    import datetime
    from time import perf_counter

    ########################################
    ###### Intitializing the Objects ########
    ########################################
    fake = Faker()
    blood_count = Counter()

    start = perf_counter()
    ########################################
    ######## Creating the Profiles ##########
    ########################################
    cnt = 10000
    for c in range(cnt):
        globals()['profile' + str(c)] = fake.profile()

    lat = 0
    lng = 0
    dob = datetime.date(datetime.datetime.now().year,
                        datetime.datetime.now().month, datetime.datetime.now().day)
    sumdob = datetime.timedelta(0)
    today = datetime.date(2021, 1, 1)

    for c in range(cnt):
        blood_count.update([(globals()['profile' + str(c)]).get('blood_group')])

        lat += (globals()['profile' + str(c)]).get('current_location')[0]
        lng += (globals()['profile' + str(c)]).get('current_location')[1]
        dob = min(dob, (globals()['profile' + str(c)]).get('birthdate'))
        sumdob += today - (globals()['profile' + str(c)]).get('birthdate')

    lat = lat/cnt
    lng = lng/cnt

    sumdob = sumdob.days / cnt
    avg_year = sumdob // 365
    avg_mnt = (sumdob - avg_year * 365) // 30
    avg_day = (sumdob - avg_year * 365 - avg_mnt*30)

    max_age = (today - dob).days
    max_year = max_age // 365
    max_mnt = (max_age - max_year * 365) // 30
    max_day = (max_age - max_year * 365 - max_mnt*30)

    common_bt = blood_count.most_common(1)[0][0]
    common_bt_cnt = blood_count.most_common(1)[0][1]
```

```
    print(
        f'The most common blood type is {common_bt} with {common_bt_cnt} counts')
    print(f'Avg. Age is {avg_year} Years, {avg_mnt} Months, and {avg_day} Days')
    print(f'Oldest. Age is {max_year} Years, {max_mnt} Months, and {max_day} Days')
    print(f'Mean location is ({lat}, {lng})')
    end = perf_counter()

    total_elapsed = end - start
    print(f'Time {total_elapsed}')

    return {'count': cnt, 'common_bt_cnt': common_bt_cnt, 'avg_year': avg_year, 'av
```

In [7]:

```
a = profile_dict()
```

```
The most common blood type is A+ with 1299 counts
Avg. Age is 57.0 Years, 1.0 Months, and 24.15100000000166 Days
Oldest. Age is 115 Years, 6 Months, and 24 Days
Mean location is (-0.14151207115, 2.1849021149)
Time 11.950717317999988
```

Test Cases

In [8]:

```
def test_profile_dict():
    a = profile_dict()
    assert a['count'] == 10000, "Number of profiles must be 10000!"
    assert a['common_bt_cnt'] <= a['count'], "Max blood type count exceeds number o
    assert [True if a['max_year'] > a['avg_year'] else (True if a['max_mnt'] > a['a
        True if a['max_day'] > a['avg_day'] else False))], "How can the max age be
    assert len(a.keys()) >= 8, 'Insufficient number of outputs in your code!'
    assert sum(number < 0 for number in [a['max_year'], a['max_mnt'], a['max_day'],
                                         a['avg_year'], a['avg_mnt'], a['avg_day']]
    assert profile_dict.__doc__, "Your function must have a docstring"

    with Capturing() as output:
        profile_dict()
    assert any(["Time" in o for o in output]), "How will you compare the time with
```

In [9]:

```
test_profile_dict()
```

```
The most common blood type is O+ with 1275 counts
Avg. Age is 57.0 Years, 10.0 Months, and 19.32170000000042 Days
Oldest. Age is 115 Years, 6 Months, and 26 Days
Mean location is (0.12567667875, 1.4052466025)
Time 10.950223064000056
```

Question 3

In [10]:

```
def company_fn():
    '''
    A function to generate profiles of 100 companies. The Stock values are
    intialized using random values (and so are the fluctuations). Low was not expli
    asked, but was hinted by the instructor while presenting test cases.
    '''
    ########################################
    ##############  Imports  ###############
    ########################################
    from faker import Faker
    from collections import namedtuple
    import random

    ########################################
    ###### Intitializing the Objects ########
    ########################################
    fake = Faker()


    ########################################
    ####### Defining the NamedTuple #########
    ########################################
    company = namedtuple(
        'company', ['name', 'symbol', 'open', 'high', 'low', 'close'])
    company.__doc__ = "Company stock profile with current market trend values"

    ########################################
    ######## Creating the Profiles ##########
    ########################################
    cnt = 100
    for c in range(cnt):
        comp_name = fake.company()
        comp_symb = comp_name
        open = random.randint(80, 950)
        fluctuations = [open * random.uniform(0.5, 1.5) for _ in range(48)]
        close = fluctuations[-1]
        high = max(fluctuations)
        low = min(fluctuations)
        globals()['company' + str(c)] = company(comp_name,
                                                comp_symb, open, high, low, close)

    ########################################
    ######## Analyzing the Profiles #########
    ########################################
    stock_open, stock_high, stock_low, stock_close = 0, 0, 0, 0
    weights = [random.uniform(0.1, 0.9) for _ in range(cnt)]
    norm_wts = [x/sum(weights) for x in weights]
    for c in range(cnt):
        stock_open += (globals()['company' + str(c)]).open * norm_wts[c]
        stock_high += (globals()['company' + str(c)]).high * norm_wts[c]
        stock_low += (globals()['company' + str(c)]).low * norm_wts[c]
        stock_close += (globals()['company' + str(c)]).close * norm_wts[c]

    ########################################
    ####### Determination of High/Low #######
    ########################################
    stock_high = stock_close if stock_close > stock_high else stock_high
    stock_low = stock_close if stock_close < stock_low else stock_low
```

```
    stock_open, stock_high, stock_low, stock_close = round(stock_open, 2), round(
        stock_high, 2), round(stock_low, 2), round(stock_close, 2)

    print(f'Stock opened at:{stock_open}')
    print(f'Stock highest:{stock_high}')
    print(f'Stock lowest:{stock_low}')
    print(f'Stock closed at:{stock_close}')

    return {'cmp_cnt': cnt, 'stock_open': stock_open, 'stock_high': stock_high, 'st
```

In [11]:

```
def test_company_fn():
    a = company_fn()
    assert a['cmp_cnt'] == 100, "Number of companies must be 100!"
    assert a['stock_high'] >= a['stock_low'], "Stock High cannot be less than Stock
    assert a['stock_high'] >= a['stock_open'], "Stock High cannot be less than Stoc
    assert a['stock_high'] >= a['stock_close'], "Stock High cannot be less than Sto
    assert a['stock_high'] < 100 * a['stock_low'], "Stock cannot vary too much"
    assert len(a.keys()) >= 5, 'Insufficient number of outputs in your code!'
    assert company_fn.__doc__, "Your function must have a docstring"

    with Capturing() as output:
        company_fn()
    assert any(["Stock open" in o for o in output]), "You must report the Stock Ope
    assert any(["Stock high" in o for o in output]), "You must report the Stock Hig
    assert any(["Stock close" in o for o in output]), "You must report the Stock En
```

In [12]:

```
test_company_fn()
```

```
Stock opened at:509.9
Stock highest:753.38
Stock lowest:265.89
Stock closed at:485.94
```