# Lab 2B: Functional programming, and declarative patterns

**Student:** lensc874 (Lennart Schilling)

**Student:** thiqu264 (Thijs Quast)

Disclaimer: Functional programming in Python does not always lead to the fastest possible code, and is often not considered the *pythonic* approach. However, functional programming is the basis for many concurrent systems (the MapReduce programming model which many big data systems, e.g. Hadoop, relies on gets its name from the *map* and *reduce* functions mentioned below). Python is a multi-paradigm language, and functional programming is one of the main paradigms one can use. To understand how and when to do this, it is necessary to do things in a non-*pythonic* way in order to cover the basics.

## General instructions

In this lab there are some general rules you should keep in mind to make sure you are on the correct path in your solutions.

#### Rules

1. You are not allowed to use `while` or `for` statements unless this is explicitly allowed in the task.
2. You are not allowed to use global variables (other than for functions defined in the global environment).
3. Code stubs should be viewed as fixed, you are only allowed to add code, the only code you are allowed to change is `pass` statements, which you should remove.
4. You should refrain from using the `list` datatype unless otherwise specified and instead use `tuple`. One of the strengths of functional programming is its focus on immutable data types (this is why functional programming and concurrency goes so well together). Incidentally, one might find speedups when using the immutable tuples instead of lists.

#### Advice

1. Avoid local variables unless you are certain they are necessary, in most cases you won't need to use local variables. (altermatively, use local variables to your hearts content, but when your solution works, try to eliminate them, you should be able to eliminate most of them, over time, you might find that you don't need them.)

# 2 Recursion

As an introduction to linear recursion, read the introductory note on the course webpage. This might help explain terms that you may not know (even if the concept is previously known).

## 2.1 Linear recursion

a) Write a recursive function `sum_even(n)` that takes a natural number $n \geq 0$ and returns the sum of all even numbers $0, \ldots, n$. It should be linear-recursive with delayed computations.

In [120]:

```
def sum_even(n):
    if n < 0:
        raise ValueError("n must be >= 0.")
    if not n:
        return n
    if n%2 != 0:
        return sum_even(n-1)
    else:
        return sum_even(n-1) + n
sum_even(100)
```

Out[120]:

```
2550
```

b) Write `sum_even_it(n)` according to the same specification. In this case, the solution should be tail recursive.

In [121]:

```
def sum_even_it(n, res=0):
    if n < 0:
        raise ValueError("n must be >= 0.")
    if n <= 1:
        return res
    if n%2 != 0:
        return sum_even_it(n-1, res)
    else:
        return sum_even_it(n-1, res + n)
sum_even_it(100)
```

Out[121]:

```
2550
```

c) We can of course express this in a declarative and Pythonic way, which is non-recursive. Write a function `sum_even_py` which returns the same result as above, but using comprehension or filter/map/reduce construct.

In [122]:

```
def sum_even_py(n):
    return(sum(filter(lambda x: x%2 == 0, range(0,n+1))))
sum_even_py(100)
```

Out[122]:

```
2550
```

# 2.2 Double/tree recursion

Sometimes we might find ourselves with branching structures, where there are several "smaller" cases to recurse over. This might for instance be the case when we have trees, lists-within-lists or the like.

[Note: In the tasks below, it might be helpful to gain an understanding of `isinstance`. See the documentation!]

a) One common use of recursion is to traverse recursive data structures. One exercise might be to *flatten* nested lists or tuples. This is relatively simple with only one level of nesting, or when the structure follows a strict pattern, but for arbitrary nested sequences, a recursive approach is more natural. Implement a recursive function `myflatten` which can take an arbitrary structure of nested tuples and flattens it (in the sense of returning a new non-nested tuple with the same elements in the same order).

In [123]:

```
def myflatten(seq):
    if seq == ():
        newsequence = ()
        return newsequence
    elif isinstance(seq[0], tuple):
        newsequence = myflatten(seq[0]) + myflatten(seq[1:])
        return newsequence
    else:
        newsequence = seq[:1] + myflatten(seq[1:])
        return newsequence

def test_myflatten():
    tests = (
     ((), (), "the empty tuple"),
     ((1,2,3), (1,2,3), "flat tuples invariant under flattening"),
     ( (1, (2), 3, (4, 5, (6), 7), 8), (1, 2, 3, 4, 5, 6, 7, 8), "Arbitrarily nested tu
    )
    for arg, expected_output, error_msg in tests:
        assert myflatten(arg) == expected_output, error_msg

test_myflatten()

# Testing.
empty = ((1,2), (1,2), 1)
myflatten(empty)
```

Out[123]:

```
(1, 2, 1, 2, 1)
```

b) Implement a function `exists_in(e, seq)` which returns `True` if the element `e` exists somewhere in the tuple `seq` (and `False` otherwise). `seq` might be nested and contain tuples-within-tuples.

In [124]:

```
def exists_in(e, seq):
    return e in myflatten(seq)
# Testing.
exists_in(20, empty)
```

Out[124]:

```
False
```

c) Write a few representative test cases, as in lab 2A.

In [125]:

```
def test_find_anywhere():

    tests = (
        ( (20, (1, 2, (3, 20), 2)), True, "test_find_anywhere should return true if se
        ( (19, (1, 2, (3, 20), 2)), False, "test_find_anywhere should return false if 
    )

    for args, output, error_msg in tests:
        assert exists_in(*args) == output, error_msg

    print("--- test_find_anywhere finished successfully")

test_find_anywhere()
```

```
--- test_find_anywhere finished successfully
```

d) One of the most famous recursive functions is the Quicksort function (https://en.wikipedia.org/wiki/Quicksort) (https://en.wikipedia.org/wiki/Quicksort)). It allows us to sort a sequence, with repeated values, in (amortized) log-linear time and with a logarithmic number of recursive calls. We will start by implementing Quicksort for a tuple of numbers.

Note that Wikipedia illustrates a more advanced *in-place* version of Quicksort, with a more advanced partition function. For the purposes of this assignment, you can simply pass a new tuple or generator to each recursive call to quicksort. You may use eg *filter* or a comprehension to create the inputs.

In [126]:

```
from random import sample, choice

def quicksort(seq):
    if seq:
        # Choosing first element of sequence as pivot.
        pivot = seq[0]
        # Identifying elements smaller than pivot.
        smaller = [i for i in seq[1:] if i < pivot]
        # Identifying elements larger than pivot.
        larger = [i for i in seq[1:] if i >= pivot]
        # Returning sorted seq.
        return quicksort(smaller) + [pivot] + quicksort(larger)
    # If seq is empty, returning empty seq.
    else:
        return seq

a = tuple(sample(range(1000,2000), 1000))
print(a)
b = quicksort(a)
print(b)
```

```
(1988, 1839, 1806, 1947, 1946, 1393, 1443, 1528, 1171, 1125, 1564, 1269, 120
8, 1985, 1399, 1067, 1788, 1143, 1851, 1974, 1356, 1679, 1368, 1980, 1997, 1
746, 1583, 1520, 1602, 1560, 1578, 1934, 1588, 1636, 1918, 1198, 1149, 1188,
1181, 1306, 1534, 1584, 1523, 1133, 1202, 1177, 1405, 1065, 1688, 1090, 174
7, 1846, 1727, 1303, 1146, 1039, 1204, 1967, 1514, 1721, 1633, 1079, 1277, 1
694, 1293, 1282, 1513, 1435, 1505, 1833, 1730, 1199, 1407, 1500, 1346, 1522,
1665, 1384, 1497, 1790, 1643, 1398, 1812, 1722, 1548, 1167, 1256, 1223, 190
1, 1259, 1991, 1634, 1252, 1043, 1364, 1163, 1644, 1270, 1930, 1795, 1477, 1
233, 1042, 1064, 1330, 1192, 1782, 1909, 1873, 1656, 1888, 1438, 1756, 1285,
1134, 1475, 1483, 1811, 1789, 1908, 1732, 1543, 1263, 1993, 1278, 1288, 169
7, 1854, 1059, 1237, 1168, 1337, 1104, 1212, 1592, 1206, 1249, 1299, 1899, 1
862, 1301, 1742, 1429, 1253, 1071, 1325, 1108, 1474, 1348, 1442, 1808, 1868,
1995, 1165, 1182, 1814, 1973, 1637, 1447, 1417, 1228, 1589, 1135, 1381, 123
8, 1078, 1600, 1273, 1130, 1103, 1507, 1294, 1107, 1964, 1879, 1087, 1489, 1
936, 1567, 1866, 1162, 1852, 1116, 1159, 1030, 1810, 1008, 1482, 1229, 1005,
1418, 1311, 1341, 1003, 1469, 1329, 1131, 1609, 1276, 1183, 1693, 1327, 170
4, 1787, 1938, 1915, 1423, 1628, 1531, 1793, 1802, 1996, 1710, 1013, 1117, 1
515, 1815, 1215, 1111, 1998, 1371, 1809, 1701, 1069, 1495, 1535, 1937, 1759,
1241, 1829, 1905, 1390, 1731, 1770, 1283, 1245, 1152, 1524, 1227, 1519, 175
4, 1415, 1677, 1550, 1616, 1914, 1209, 1978, 1333, 1608, 1867, 1232, 1639, 1
300, 1296, 1230, 1314, 1648, 1569, 1473, 1668, 1274, 1703, 1266, 1577, 1786,
1004, 1629, 1652, 1394, 1882, 1092, 1940, 1603, 1540, 1700, 1619, 1445, 104
7, 1350, 1180, 1034, 1682, 1240, 1555, 1698, 1897, 1614, 1856, 1460, 1551, 1
613, 1191, 1402, 1312, 1054, 1169, 1062, 1305, 1872, 1763, 1885, 1984, 1374,
1612, 1955, 1831, 1157, 1401, 1095, 1518, 1411, 1642, 1502, 1813, 1210, 182
1, 1807, 1822, 1436, 1464, 1018, 1557, 1281, 1451, 1068, 1403, 1313, 1225, 1
667, 1630, 1132, 1058, 1431, 1605, 1925, 1920, 1723, 1038, 1702, 1758, 1760,
1354, 1771, 1323, 1388, 1041, 1427, 1834, 1142, 1958, 1681, 1419, 1565, 112
2, 1728, 1468, 1160, 1440, 1014, 1336, 1044, 1941, 1211, 1286, 1658, 1570, 1
780, 1164, 1850, 1382, 1467, 1655, 1586, 1113, 1310, 1707, 1174, 1373, 1367,
1255, 1512, 1593, 1154, 1127, 1954, 1504, 1889, 1753, 1895, 1456, 1960, 108
2, 1712, 1391, 1830, 1585, 1179, 1061, 1433, 1772, 1638, 1148, 1377, 1738, 1
714, 1692, 1063, 1100, 1525, 1622, 1437, 1990, 1962, 1762, 1496, 1969, 1331,
1683, 1425, 1430, 1322, 1421, 1527, 1726, 1324, 1289, 1999, 1216, 1781, 129
8, 1887, 1205, 1883, 1362, 1375, 1457, 1891, 1380, 1392, 1663, 1855, 1724, 1
953, 1884, 1870, 1860, 1604, 1553, 1890, 1740, 1028, 1929, 1383, 1109, 1820,
1875, 1917, 1654, 1076, 1049, 1040, 1849, 1404, 1326, 1863, 1345, 1532, 135
```

```
2, 1173, 1158, 1070, 1251, 1741, 1328, 1023, 1153, 1572, 1948, 1114, 1646, 1
650, 1711, 1615, 1706, 1239, 1778, 1444, 1511, 1824, 1949, 1309, 1379, 1581,
1708, 1618, 1594, 1439, 1881, 1910, 1412, 1684, 1452, 1344, 1002, 1319, 157
6, 1287, 1036, 1376, 1450, 1861, 1549, 1761, 1434, 1819, 1571, 1213, 1597, 1
184, 1007, 1089, 1832, 1492, 1871, 1911, 1315, 1735, 1779, 1193, 1526, 1959,
1562, 1123, 1803, 1792, 1695, 1156, 1919, 1397, 1744, 1847, 1466, 1755, 189
4, 1748, 1775, 1189, 1651, 1051, 1085, 1626, 1632, 1358, 1291, 1661, 1052, 1
385, 1357, 1340, 1971, 1720, 1031, 1370, 1033, 1243, 1666, 1595, 1539, 1590,
1641, 1050, 1491, 1267, 1021, 1361, 1487, 1236, 1257, 1992, 1271, 1783, 131
8, 1552, 1785, 1118, 1119, 1558, 1186, 1580, 1396, 1449, 1624, 1880, 1360, 1
503, 1378, 1175, 1481, 1739, 1077, 1510, 1422, 1465, 1935, 1951, 1408, 1332,
1185, 1546, 1647, 1979, 1968, 1260, 1317, 1797, 1843, 1265, 1757, 1923, 152
9, 1203, 1972, 1197, 1190, 1840, 1366, 1671, 1800, 1620, 1308, 1347, 1983, 1
432, 1201, 1359, 1025, 1387, 1074, 1480, 1537, 1556, 1944, 1627, 1072, 1750,
1389, 1207, 1574, 1321, 1096, 1355, 1351, 1945, 1931, 1837, 1877, 1939, 122
6, 1970, 1024, 1729, 1878, 1414, 1563, 1893, 1334, 1470, 1484, 1904, 1136, 1
776, 1015, 1678, 1106, 1099, 1075, 1660, 1912, 1395, 1369, 1428, 1976, 1625,
1010, 1053, 1965, 1705, 1664, 1097, 1777, 1874, 1913, 1922, 1080, 1900, 167
6, 1472, 1145, 1950, 1224, 1566, 1093, 1737, 1110, 1151, 1765, 1696, 1486, 1
907, 1178, 1302, 1258, 1826, 1987, 1751, 1796, 1835, 1752, 1994, 1903, 1773,
1853, 1876, 1141, 1599, 1672, 1718, 1575, 1490, 1799, 1279, 1262, 1842, 194
2, 1307, 1963, 1458, 1386, 1924, 1200, 1545, 1764, 1137, 1767, 1733, 1685, 1
081, 1686, 1121, 1716, 1892, 1544, 1138, 1086, 1268, 1234, 1928, 1554, 1231,
1859, 1932, 1083, 1957, 1046, 1264, 1155, 1066, 1048, 1316, 1745, 1818, 112
6, 1244, 1791, 1424, 1112, 1172, 1865, 1284, 1446, 1804, 1400, 1485, 1582, 1
416, 1825, 1645, 1016, 1926, 1498, 1886, 1687, 1426, 1561, 1517, 1801, 1410,
1943, 1001, 1670, 1952, 1631, 1463, 1219, 1057, 1029, 1640, 1032, 1827, 134
9, 1857, 1598, 1338, 1674, 1506, 1794, 1339, 1246, 1521, 1898, 1623, 1836, 1
478, 1011, 1591, 1902, 1166, 1536, 1045, 1541, 1845, 1817, 1533, 1441, 1956,
1027, 1494, 1147, 1966, 1657, 1295, 1805, 1455, 1479, 1798, 1530, 1547, 122
2, 1098, 1516, 1084, 1196, 1304, 1828, 1161, 1448, 1187, 1774, 1420, 1921, 1
579, 1690, 1715, 1139, 1675, 1275, 1653, 1471, 1906, 1981, 1406, 1459, 1848,
1094, 1986, 1823, 1816, 1621, 1559, 1235, 1659, 1140, 1607, 1022, 1454, 170
9, 1673, 1587, 1120, 1699, 1610, 1088, 1689, 1060, 1499, 1568, 1144, 1916, 1
242, 1927, 1743, 1680, 1037, 1124, 1000, 1462, 1933, 1297, 1713, 1869, 1105,
1784, 1542, 1220, 1214, 1292, 1343, 1611, 1290, 1150, 1091, 1982, 1006, 183
8, 1365, 1734, 1989, 1254, 1669, 1508, 1617, 1858, 1717, 1020, 1975, 1247, 1
102, 1725, 1363, 1221, 1017, 1176, 1101, 1596, 1493, 1248, 1073, 1413, 1606,
1766, 1977, 1844, 1194, 1768, 1261, 1453, 1896, 1280, 1501, 1335, 1749, 117
0, 1129, 1115, 1195, 1488, 1961, 1461, 1635, 1026, 1012, 1662, 1019, 1372, 1
719, 1864, 1009, 1218, 1769, 1691, 1509, 1409, 1217, 1601, 1035, 1250, 1055,
1573, 1056, 1476, 1649, 1342, 1128, 1841, 1736, 1320, 1538, 1353, 1272)
[1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011, 101
2, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1
025, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037,
1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 105
0, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059, 1060, 1061, 1062, 1
063, 1064, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1072, 1073, 1074, 1075,
1076, 1077, 1078, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1087, 108
8, 1089, 1090, 1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1100, 1
101, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113,
1114, 1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 112
6, 1127, 1128, 1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1
139, 1140, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151,
1152, 1153, 1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 116
4, 1165, 1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1
177, 1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188, 1189,
1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1201, 120
2, 1203, 1204, 1205, 1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1
215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227,
1228, 1229, 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1238, 1239, 124
```

```
0, 1241, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1249, 1250, 1251, 1252, 1
253, 1254, 1255, 1256, 1257, 1258, 1259, 1260, 1261, 1262, 1263, 1264, 1265,
1266, 1267, 1268, 1269, 1270, 1271, 1272, 1273, 1274, 1275, 1276, 1277, 127
8, 1279, 1280, 1281, 1282, 1283, 1284, 1285, 1286, 1287, 1288, 1289, 1290, 1
291, 1292, 1293, 1294, 1295, 1296, 1297, 1298, 1299, 1300, 1301, 1302, 1303,
1304, 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315, 131
6, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1328, 1
329, 1330, 1331, 1332, 1333, 1334, 1335, 1336, 1337, 1338, 1339, 1340, 1341,
1342, 1343, 1344, 1345, 1346, 1347, 1348, 1349, 1350, 1351, 1352, 1353, 135
4, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1
367, 1368, 1369, 1370, 1371, 1372, 1373, 1374, 1375, 1376, 1377, 1378, 1379,
1380, 1381, 1382, 1383, 1384, 1385, 1386, 1387, 1388, 1389, 1390, 1391, 139
2, 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1
405, 1406, 1407, 1408, 1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417,
1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425, 1426, 1427, 1428, 1429, 143
0, 1431, 1432, 1433, 1434, 1435, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1
443, 1444, 1445, 1446, 1447, 1448, 1449, 1450, 1451, 1452, 1453, 1454, 1455,
1456, 1457, 1458, 1459, 1460, 1461, 1462, 1463, 1464, 1465, 1466, 1467, 146
8, 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1476, 1477, 1478, 1479, 1480, 1
481, 1482, 1483, 1484, 1485, 1486, 1487, 1488, 1489, 1490, 1491, 1492, 1493,
1494, 1495, 1496, 1497, 1498, 1499, 1500, 1501, 1502, 1503, 1504, 1505, 150
6, 1507, 1508, 1509, 1510, 1511, 1512, 1513, 1514, 1515, 1516, 1517, 1518, 1
519, 1520, 1521, 1522, 1523, 1524, 1525, 1526, 1527, 1528, 1529, 1530, 1531,
1532, 1533, 1534, 1535, 1536, 1537, 1538, 1539, 1540, 1541, 1542, 1543, 154
4, 1545, 1546, 1547, 1548, 1549, 1550, 1551, 1552, 1553, 1554, 1555, 1556, 1
557, 1558, 1559, 1560, 1561, 1562, 1563, 1564, 1565, 1566, 1567, 1568, 1569,
1570, 1571, 1572, 1573, 1574, 1575, 1576, 1577, 1578, 1579, 1580, 1581, 158
2, 1583, 1584, 1585, 1586, 1587, 1588, 1589, 1590, 1591, 1592, 1593, 1594, 1
595, 1596, 1597, 1598, 1599, 1600, 1601, 1602, 1603, 1604, 1605, 1606, 1607,
1608, 1609, 1610, 1611, 1612, 1613, 1614, 1615, 1616, 1617, 1618, 1619, 162
0, 1621, 1622, 1623, 1624, 1625, 1626, 1627, 1628, 1629, 1630, 1631, 1632, 1
633, 1634, 1635, 1636, 1637, 1638, 1639, 1640, 1641, 1642, 1643, 1644, 1645,
1646, 1647, 1648, 1649, 1650, 1651, 1652, 1653, 1654, 1655, 1656, 1657, 165
8, 1659, 1660, 1661, 1662, 1663, 1664, 1665, 1666, 1667, 1668, 1669, 1670, 1
671, 1672, 1673, 1674, 1675, 1676, 1677, 1678, 1679, 1680, 1681, 1682, 1683,
1684, 1685, 1686, 1687, 1688, 1689, 1690, 1691, 1692, 1693, 1694, 1695, 169
6, 1697, 1698, 1699, 1700, 1701, 1702, 1703, 1704, 1705, 1706, 1707, 1708, 1
709, 1710, 1711, 1712, 1713, 1714, 1715, 1716, 1717, 1718, 1719, 1720, 1721,
1722, 1723, 1724, 1725, 1726, 1727, 1728, 1729, 1730, 1731, 1732, 1733, 173
4, 1735, 1736, 1737, 1738, 1739, 1740, 1741, 1742, 1743, 1744, 1745, 1746, 1
747, 1748, 1749, 1750, 1751, 1752, 1753, 1754, 1755, 1756, 1757, 1758, 1759,
1760, 1761, 1762, 1763, 1764, 1765, 1766, 1767, 1768, 1769, 1770, 1771, 177
2, 1773, 1774, 1775, 1776, 1777, 1778, 1779, 1780, 1781, 1782, 1783, 1784, 1
785, 1786, 1787, 1788, 1789, 1790, 1791, 1792, 1793, 1794, 1795, 1796, 1797,
1798, 1799, 1800, 1801, 1802, 1803, 1804, 1805, 1806, 1807, 1808, 1809, 181
0, 1811, 1812, 1813, 1814, 1815, 1816, 1817, 1818, 1819, 1820, 1821, 1822, 1
823, 1824, 1825, 1826, 1827, 1828, 1829, 1830, 1831, 1832, 1833, 1834, 1835,
1836, 1837, 1838, 1839, 1840, 1841, 1842, 1843, 1844, 1845, 1846, 1847, 184
8, 1849, 1850, 1851, 1852, 1853, 1854, 1855, 1856, 1857, 1858, 1859, 1860, 1
861, 1862, 1863, 1864, 1865, 1866, 1867, 1868, 1869, 1870, 1871, 1872, 1873,
1874, 1875, 1876, 1877, 1878, 1879, 1880, 1881, 1882, 1883, 1884, 1885, 188
6, 1887, 1888, 1889, 1890, 1891, 1892, 1893, 1894, 1895, 1896, 1897, 1898, 1
899, 1900, 1901, 1902, 1903, 1904, 1905, 1906, 1907, 1908, 1909, 1910, 1911,
1912, 1913, 1914, 1915, 1916, 1917, 1918, 1919, 1920, 1921, 1922, 1923, 192
4, 1925, 1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933, 1934, 1935, 1936, 1
937, 1938, 1939, 1940, 1941, 1942, 1943, 1944, 1945, 1946, 1947, 1948, 1949,
1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 196
2, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1
975, 1976, 1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987,
1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999]
```

# 3 Higher order functions (HOF)

A *higher-order function* is a function which operates on other functions. What this means exactly is disputed, but we will call any function which returns a function or takes a function as an argument a higher-order function. (Conversely, a function neither taking another function as input nor returning a function we will refer to as a *first-order function*)

In R you have encountered these when, for instance, using the `apply` family of functions, which are all versions of what is called a `map` function in functional programming (see below).

When using higher-order functions, it is often useful to create simple anonymous functions at the place in the code where they are used, rather than defining a new named function in one place only to call it in a single other place. In R, all functions are created in this way with the `function` keyword, but they are usually assigned to global names with standard assignment ( `<-` ). Python provides similar functionality using the `lambda` keyword (name inspired by Alonzo Church's $\lambda$-calculus (https://www.youtube.com/watch?v=eis11j_iGMs) which has inspired much of functional programming) with which we can create anonymous functions. Of course, we can also pass named functions to higher-order functions, which is usually the case when the function is predefined, general enough to be used in more than one place, or complex enough to warrant separate definition and documentation for the sake of clarity.

## 3.1 The three standard functions `map`, `reduce` and `filter`

There are three standard cases which are widely applicable and many other higher-order functions are special cases or combinations of these. They are: `map` , apply a function on each element in a sequence, `filter` , keep (or conversely, remove) elements from a sequence according to some condition, and `reduce` , combine the elements in a sequence. The `map` function takes a sequence and a function (usually of 1 parameter) which is to be applied to each element of the sequence and might return anything, this function is assumed not to have side effects. The `filter` function takes a function (usually of 1 parameter) which returns a boolean value used to indicate which elements are to be kept. The `reduce` function takes a function (usually of 2 parameters) which is used to combine the elements in the sequence.

In Python, `map` and `filter` are standard built-in functions. Since Python 3, the `reduce` function needs to be imported from the `functools` module.

Many more advanced functions, of any order, can be created by combining these three higher-order functions.

A note from last year: usually, the `reduce` function is more difficult to grasp than `map` and `filter` but I found this blog-post by André Burgaud to be a nice introduction to `reduce` . Note that Burgaud talks about the more general *fold* concept rather than `reduce` , which is a special case of fold often called *left fold* (this is covered in more detail in the post). https://www.burgaud.com/foldl-foldr-python/ (https://www.burgaud.com/foldl-foldr-python/)

a) Implement a function `mysum` which computes the sum of a list or tuple of numbers using the reduce function and a lambda function.

In [127]:

```
from functools import reduce

def mysum(seq):
    return reduce(lambda x, y: x+y, seq)

mysum((4, 7, 1))
```

Out[127]:

```
12
```

b) Implement a function `mylength` which uses `map` and `reduce` to compute the length of a sequence. The use of the `len` function is not allowed.

[Hint: Use `map` to convert the input to something which can easily be `reduce` :d.]

In [128]:

```
def mylength(seq):
    return reduce(lambda x, y: x+y, map(lambda x: 1, seq))

print(mylength((4, 2, 5, 2, 5)))
print(mylength("test"))
```

```
5
4
```

## 3.2 Building your own higher order functions

a) Re-implement the three basic functional helper functions `map` , `filter` and `reduce` **as purely functional recursive functions**. You may not express this as eg comprehensions; the task is to practice figuring out this type of logic.

Note that the built-in versions of these functions work on multiple sequences of equal length if supplied, however, you can assume a single sequence as second parameter, i.e. you can also skip the third parameter to reduce.

In [129]:

```
def mymap(f, seq):
    # After a while, empty sequence will be put into function. Then process needs to be
    if seq == ():
        return ()
    # If sequence is an integer, function shall be applied.
    if isinstance(seq, int):
        return (f(seq),)
    # If sequence is not an integer (but in this case a tuple instead), function will l
    # twice with first element and other elements as input.
    else:
        return mymap(f, seq[0]) + mymap(f, seq[1:])

mymap(lambda x:x**2, tuple(range(10)))
```

Out[129]:

```
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
```

In [130]:

```
def myfilter(f, seq):
    # After a while, empty sequence will be put into function. Then process needs to be
    if seq == ():
        return ()
    # If input sequence is an integer, sequence shall be returned if f(seq) == true.
    if isinstance(seq, int):
        if f(seq):
            return (seq,)
        else:
            return()
    # If sequence is not an integer (but in this case a tuple instead), function will l
    # twice with first element and other elements as input.
    else:
        return myfilter(f, seq[0]) + myfilter(f, seq[1:])

myfilter(lambda x:x%2==0, tuple(range(10)))
```

Out[130]:

```
(0, 2, 4, 6, 8)
```

You might note the similarities with how you implemented `sum_even`.

In [131]:

```
def myreduce(f, seq):
    if not seq:
        return seq
    elif not isinstance(seq, (tuple, list)):
        return seq
    else:
        one = myreduce(f, seq[0])
        two = myreduce(f, seq[1:])
        if two:
            return f(one, two)
        else:
            return one

myreduce(lambda x, y: x*y, tuple(range(1,5)))
```

Out[131]:

24

## 3.3 Returning functions

The previous section covered functions which take other functions as input, but what about the opposite, functions returning functions as output?

a) Function composition is a common in both maths and programming. Write a function `compose` which takes two functions, $f$ and $g$, and produces the *composite* function $f \circ g$, where $(f \circ g)(x) \Leftrightarrow f(g(x))$. Example use is given below.

In [132]:

```
from statistics import stdev, mean

def compose(f, g):
    return lambda a: f(g(a))

def myscale(vals):
    return [x/stdev(vals) for x in vals]

def myshift(vals):
    return [x-mean(vals) for x in vals]

standardize = compose(myscale, myshift)

print(standardize(range(-3, 8)))
```

```
[-1.507556722888818, -1.2060453783110545, -0.9045340337332909, -0.6030226891
555273, -0.30151134457776363, 0.0, 0.30151134457776363, 0.6030226891555273,
0.9045340337332909, 1.2060453783110545, 1.507556722888818]
```

b) Create a function `composition(*funs)` which takes a non-empty sequence of functions of one argument and returns their sequential composition. That is $composition(f_0, f_1, \ldots, f_n) = f_0 \circ f_1 \circ \ldots \circ f_n$. (The question of if $f \circ g \circ h$ should be read $f \circ (g \circ h)$ or $(f \circ g) \circ h$ is perfectly valid, but they turn out to be the same. That is, $\circ$ is associative.)

In [133]:

```
def compose(*funs):
    def compose_inner(f, g):
        return lambda x: f(g(x))
    return reduce(compose_inner, funs, lambda x: x)
```

Hint: Don't remember what can be found in `*funs` ? Print it! Don't know how the values should be combined? Write out some simple example on paper.

Note: This task demonstrates the generality of our constructs. Previously we worked with sequences of numbers and the like. Now we lift this to the level of working with functions as values, and instead of using combinators which work on numbers, we use function combinators in conjunction with our known patterns.

## Voluntary task: pipelining

When doing data analysis, one very important part is pre-processing. Often, data goes through a number of steps of preprocessing, sometimes called a pipeline. The function composition example above can be seen as a special case of such a pipeline for only two functions. By clever use of higher order functions, we can build a pipeline function which takes a list or tuple of data transforming functions and creates a function which applies these sequentially. Construct such a function called `make_pipeline` . In order to focus on the primary purpose of the `make_pipeline` function, we will perform a very simple set of transformations, increment each value by 1, take the absolute value, and then take the square root. Usage example and code for the `inc` function is supplied below.

You may want to use functions you have defined above.

In [134]:

```
from functools import reduce, partial
from math import sqrt


def make_pipeline(*funs):
    return lambda vals: ???

# We can even drop the lambda vals : bit, using partial
# evaluation (see the help for functools.partial!)

def inc(x):
    return x+1

pipeline = make_pipeline(inc, abs, sqrt)

tuple(pipeline(range(-5,5)))
```

```
  File "<ipython-input-134-c97bccd4fa25>", line 6
    return lambda vals: ???
                        ^
SyntaxError: invalid syntax
```

# 4. Simple declarative Pythonic patterns (involing HOF)

a) As preparation, create a named tuple type "coord" which has fields `x` and `y`.

In [135]:

```
# Add the requisite import statement here
from collections import namedtuple

coord = namedtuple('coord', ['x', 'y'])
five_three = coord(5,3)
assert five_three.x == 5, "first element is the x coordinate"
assert five_three.y == 3, "the second element is the y coordinate"
```

b) Generate a $10^7$ random coordinates, with $x$ and $y$ coordinates drawn uniformly from [-1000,1000]. Save the tuple of those with $x + y > 0$ as `rnd_coords`. How many are there?

In [136]:

```
from random import uniform
def generate_coordinates(n):
    coordinate = coord(uniform(-1000, 1000), uniform(-1000, 1000))
    return coordinate
```

In [137]:

```
all_coordinates = list(map(generate_coordinates, range(10**7)))
```

In [138]:

```
len(all_coordinates)
```

Out[138]:

```
10000000
```

In [139]:

```
rnd_coords = tuple(filter(lambda coordinate: (coordinate.x + coordinate.y) >0, all_coo
```

In [140]:

```
len(rnd_coords)
```

Out[140]:

```
5000213
```

In [141]:

```
rnd_coords_copy = rnd_coords

```

[Note: If this takes a while, you might want to consider when the elements are generated and saved.]

**Before having solved the tasks below, consider setting `coords` to a smaller set (eg generate $10^3$ elements instead of $10^7$ to start with).**

c) Let `sorted_rnd` be the coordinated sorted first by the `x` component and then the `y`. Use a built-in Python sorting function. Do you need any extra parameters? Why? Why not? How would you find out where the order comes from (and might it be consistent but useless, eg sorting the elements by memory location)?

In [142]:

```
def first_element(element):
    return element[0]
def second_element(element):
    return element[1]
# Sorting by x.
rnd_coords_list = list(rnd_coords)
rnd_coords_copy = list(rnd_coords_copy)
rnd_coords_list.sort(key = first_element)
tuple(rnd_coords_list)
```

Out[142]:

```
(coord(x=-999.4511962304547, y=999.9023214647179),
 coord(x=-999.3411150358784, y=999.6978107352134),
 coord(x=-998.9680380773256, y=999.7919491394064),
 coord(x=-998.5700244301883, y=998.6823539111513),
 coord(x=-998.2069988773792, y=998.2637637771002),
 coord(x=-997.8267141736175, y=999.7132005549208),
 coord(x=-997.7587327889319, y=999.4296157308047),
 coord(x=-997.7375234494273, y=998.5303092924839),
 coord(x=-997.7151628359229, y=998.3630664939242),
 coord(x=-997.5948766945455, y=999.1368241280268),
 coord(x=-996.9786411184535, y=996.9920312926085),
 coord(x=-996.7853289422885, y=996.8671170708317),
 coord(x=-996.7828306763406, y=998.481624319109),
 coord(x=-996.6640623168435, y=997.4913061917568),
 coord(x=-996.6611499202977, y=999.8151538972718),
 coord(x=-996.551695073527, y=998.2882767950518),
 coord(x=-996.4775941830412, y=999.5990147307077),
 coord(x=-996.4405181092926, y=998.6860740501079),
```

In [143]:

```
# Sorting by y.
rnd_coords_list.sort(key = second_element)
tuple(rnd_coords_list)
```

Out[143]:

```
(coord(x=999.7625198364524, y=-999.523196288622),
 coord(x=998.8823532750098, y=-998.5679867826756),
 coord(x=998.4546472227742, y=-998.1493946936147),
 coord(x=998.4096564797283, y=-997.7752466530512),
 coord(x=999.950122752279, y=-997.6575480496001),
 coord(x=997.936033412795, y=-997.6393095864499),
 coord(x=998.1479324208779, y=-997.3577724696246),
 coord(x=998.7134280095245, y=-997.3323537417738),
 coord(x=997.4967135074219, y=-997.2258060125409),
 coord(x=997.8818263784744, y=-997.0376604978588),
 coord(x=999.2853114669433, y=-996.8089002343756),
 coord(x=999.2840994642961, y=-996.7458826551585),
 coord(x=998.592289949639, y=-996.6008863104261),
 coord(x=996.768512922212, y=-996.5079617947985),
 coord(x=999.321746144469, y=-996.4361177906073),
 coord(x=996.4792888365359, y=-996.4027670081366),
 coord(x=996.310769791148, y=-996.2839715650742),
 coord(x=999.9284803185164, y=-996.2358084084559),
```

[General words of advice:

- During testing, you might want to use a smaller data set (and then try it out at a larger set).
- You might not want to display the entire list to see if you're right all the time. Slicing out the first and last elements, say the first or last 10, might provide some hints.
- You could naturally define a function which checks that the list is in order (or performs some probabilistic sampling test), to test this.]

d) Sort the values (in the sense of returning a new sorted tuple) by their Euclidean distance to the point (5,3). Continue using a built-in Python sorting function.

In [144]:

```
import numpy
from scipy.spatial import distance
pt_5_3 = (5, 3)

```

In [145]:

```
def distance_to_point(point):
    dst = distance.euclidean(point, pt_5_3)
    return (point,dst)
```

In [146]:

```
pts_near_53 = list(map(distance_to_point, rnd_coords_list))
```

In [147]:

```
def third_element(element):
    return element[1]

pts_near_53.sort(key = third_element)
```

Note: here we customise the behaviour of a built-in function by passing it information about our intended ordering.

d) Define the function `sorted_by_distance(origo)` which takes a coordinate `origo` and returns a function which sorts the sequence by the euclidean distance to `origo`. (Ie those closest to origo come first in the list.)

In [148]:

```
def sorted_by_distance(origo):

    def distance_to_point2(coordinate):
        dst = distance.euclidean(coordinate, origo)
        return (coordinate, dst)
    return distance_to_point2

ordered_by_closeness_to_53 = sorted_by_distance(coord(5,3))
pts_near_53_2 = list(map(ordered_by_closeness_to_53, rnd_coords))
pts_near_53_2.sort(key=third_element)
assert pts_near_53 == pts_near_53_2
```

[Note: Here we extend the work above to a higher-order function, which uses the local value of `origo`. In essence, this task summarises higher order functionality - we create a closure, return a function and use a custom ordering ]

e) So far in the course, we have seen, and possibly used `enumerate`, `range`, `zip`, `map` and `filter` as declarative constructs (along with the general comprehension syntax). Now we introduce a further useful iterator construct. Construct something called `reverse_squared` which when prompted would give us the squares of elements 0,...,N *but in reverse* (that is $N^2, (N-1)^2, \ldots, 2^2, 1^2, 0^2$).

In [149]:

```
# The time it takes to run this shouldn't really depend on if you use SMALL_N or BIG_N

BIG_N = 99999999
SMALL_N = 999

N = SMALL_N    # change this to test later on

reverse_squares = list((map(lambda x: x ** 2, range(N, -1, -1))))
reverse_squares
```

Out[149]:

```
[998001,
 996004,
 994009,
 992016,
 990025,
 988036,
 986049,
 984064,
 982081,
 980100,
 978121,
 976144,
 974169,
 972196,
 970225,
 968256,
 966289,
 964324,
```

In [150]:

```
# Experimentation: copy and paste your code from above into this cell.
# This is rather crude, but we want you to to be able to trust that any
# slowness in the cell above can be found by reference to that code, not the
# profiling code below.

# Copy-pasting as it might be useful to have fresh maps.


import profile

# We cut and paste this code
BIG_N = 99999999
SMALL_N = 999

N = SMALL_N
# Look at the run time. Switching from BIG_N to SMALL_N shouldn't really matter.
# This suggests that we have quick access to elements at the end of our (squared) range

reverse_squares = list(map(lambda x: x ** 2, range(N, -1, -1))) # <<---------- Your c


profile.run("print('Did we find it? ', {} in reverse_squares)".format( N**2 ))
```

```
Did we find it?  True
         60 function calls in 0.001 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        5    0.000    0.000    0.000    0.000 :0(acquire)
        1    0.000    0.000    0.001    0.001 :0(exec)
        4    0.000    0.000    0.000    0.000 :0(getpid)
        4    0.000    0.000    0.000    0.000 :0(isinstance)
        1    0.000    0.000    0.001    0.001 :0(print)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        5    0.000    0.000    0.000    0.000 :0(urandom)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        5    0.000    0.000    0.000    0.000 iostream.py:180(schedule)
        4    0.000    0.000    0.000    0.000 iostream.py:284(_is_master_
process)
        4    0.000    0.000    0.000    0.000 iostream.py:297(_schedule_f
lush)
```

Note: once you know of the construct, this task is extremely simple. It mostly serves as a demonstration of the availability of these constructs, and how they can be combined. Also, it points to efficiency considerations when using declarative iterator constructs as opposed to fixed computed structures.

As the profiling code above suggests, where we redefine the object in every run, we do not have a purely functional construct. In that case, we wouldn't be able to exhaust the values.

[Additional reading: some additional tools are available in the `itertools` module.]

# 5 Mutating function state

A function always has access to the environment in which it was created. Usually, this means that the function can access global variables. It also means that it can access and modify local bindings from where it was created.

A closure is a function which has access to an environment which is not accessible from outside the function (but which is not destroyed when the function returns). I.e. it is a way to introduce a small measure of statefulness into functional programming. In Python, iterators and generators work much like this. However, we can use the general concept in many cases.

a) Implement a function `make_counter` which has a single parameter `n` which acts as the initial value for a counter. The function should return a function with no parameters which, when called, increments the value of `n` by 1 and returns the new value.

In [151]:

```
def make_counter(n):
    def counter_plus():
        nonlocal n
        n = n + 1
        return n
    return counter_plus

counter_A = make_counter(0)
counter_B = make_counter(15)
print("To show that the functions do not affect each others' states, consider the print
print("counter_A returns: {}".format(counter_A()))
print("counter_A returns: {}".format(counter_A()))
print("counter_B returns: {}".format(counter_B()))
print("counter_A returns: {} (was it affected by the call to counter_B?)".format(counte
```

```
To show that the functions do not affect each others' states, consider the p
rintout:
counter_A returns: 1
counter_A returns: 2
counter_B returns: 16
counter_A returns: 3 (was it affected by the call to counter_B?)
```

# 6. Use case: parametrisation

Above, we see how `sorted` can be parametrised with information about the intended order. We want to extend our `quicksort` to work the same way. We should be able to provide the way for it to tell if object A in the tuple should come before object B, or after. This is done by mapping the objects onto something where we do have an order.

a) Copy your code from the `quicksort` task above, and extend it. Call the function `quicksort_param` for parametrised, and allow a key parameter to be passed in (like in `sorted`). Note that the key function should be optional. We thus want default arguments.

In [152]:

```
from random import random

# Write quicksort_param here:
def quicksort_param(seq, f = lambda x: x):
    if seq:
        # Choosing first element of sequence as pivot.
        pivot = seq[0]
        # Identifying elements smaller than pivot.
        smaller = [i for i in seq[1:] if f(i) < f(pivot)]
        # Identifying elements larger than pivot.
        larger = [i for i in seq[1:] if f(i) >= f(pivot)]
        # Returning sorted seq.
        return quicksort_param(smaller, f) + [pivot] + quicksort_param(larger, f)
    # If seq is empty, returning empty seq.
    else:
        return seq

a = tuple(tuple(random() for i in range(3)) for j in range(10))
print(a)
b = quicksort_param(a, sum)   # Elements are three-tuples. Those with smallest sums of
print(b)
print(quicksort_param([5,2,9,200]))   # No key function provided.
```

```
((0.6978082255736705, 0.13092358715345198, 0.08535743453688494), (0.74226535
36578964, 0.8652774681053433, 0.8330465986942767), (0.24222289767437377, 0.4
000198529351654, 0.7565044282035757), (0.12016807807296215, 0.32657743040988
85, 0.34792111284839333), (0.4382148846469084, 0.16001410466915655, 0.248101
5562479003), (0.13929930100946597, 0.7755073340906281, 0.5706839076275692),
(0.2890849399728498, 0.046417312117466025, 0.6155943276366791), (0.668178699
1418044, 0.8712751842064643, 0.4379390091006765), (0.5395137678026031, 0.479
132131914226, 0.96116237680063), (0.21184898883406356, 0.3641913478339126,
0.8387200543833502))
[(0.12016807807296215, 0.3265774304098885, 0.34792111284839333), (0.43821488
46469084, 0.16001410466915655, 0.2481015562479003), (0.6978082255736705, 0.1
3092358715345198, 0.08535743453688494), (0.2890849399728498, 0.0464173121174
66025, 0.6155943276366791), (0.24222289767437377, 0.4000198529351654, 0.7565
044282035757), (0.21184898883406356, 0.3641913478339126, 0.838720054383350
2), (0.13929930100946597, 0.7755073340906281, 0.5706839076275692), (0.668178
6991418044, 0.8712751842064643, 0.4379390091006765), (0.5395137678026031, 0.
479132131914226, 0.96116237680063), (0.7422653536578964, 0.8652774681053433,
0.8330465986942767)]
[2, 5, 9, 200]
```

# Attribution

Lab by Johan Falkenjack (2018), extended and rewritten by Anders Märak Leffler (2019).