**Итератор** - объект, класс которого реализует методы \_\_iter\_\_() и \_\_next\_\_()

Написать итератор, перебирающий числа от 0 до n:

|  |
| --- |
| import unittest  class MyIter:  def \_\_init\_\_(self, n):  self.i = 0  self.n = n  def \_\_iter\_\_(self):  return self  def \_\_next\_\_(self):  if self.i < self.n:  i = self.i  self.i += 1  return i  else:  raise StopIteration()  class TestMyIter(unittest.TestCase):  def test\_run(self):  arr = tuple(MyIter(5))  self.assertEqual(arr, (0, 1, 2, 3, 4))  if \_\_name\_\_ == "\_\_main\_\_":  unittest.main() |

**Генератор** - инструмент для создания итераторов. Представляет собой функцию, содержащую в своем теле ключевое слово «yield» в том месте, где необходимо вернуть данные. Генератор хранит значения переменных и последнюю выполненную инструкцию и при вызове метода \_\_next\_\_() возобновляет работу с места остановки. Отличие генератора от итератора:

1) Генератор реализован в виде функции

2) Генератор автоматически реализует методы \_\_iter\_\_() и \_\_next\_\_()

3) Генератор автоматически сохраняет между вызовами значения локальных переменных и состояние выполнения

4) Генератор автоматически вызывает StopIteration при завершении

Написать генератор, возвращающий числа от 0 до n:

|  |
| --- |
| import unittest  def my\_gen(n):  i = 0  while i < n:  yield i  i += 1  class TestMyGen(unittest.TestCase):  def test\_run(self):  arr = tuple(my\_gen(5))  self.assertEqual(arr, (0, 1, 2, 3, 4))  if \_\_name\_\_ == "\_\_main\_\_":  unittest.main() |

Написать генератор, возвращающий n первых чисел из последовательности фибоначчи

|  |
| --- |
| import unittest  def fib(n):  a, b = 0, 1  i = 0  while i < n:  yield a  a, b = b, a + b  i += 1  class TestFib(unittest.TestCase):  def test\_run(self):  arr = tuple(fib(5))  self.assertEqual(arr, (0, 1, 1, 2, 3))  if \_\_name\_\_ == "\_\_main\_\_":  unittest.main() |

Написать генератор, возвращающий числа фибоначчи, меньшие max\_val:

|  |
| --- |
| import unittest  def fib(max\_val):  a, b = 0, 1  while a < max\_val:  yield a  a, b = b, a + b  class TestFib(unittest.TestCase):  def test\_run(self):  arr = tuple(fib(5))  self.assertEqual(arr, (0, 1, 1, 2, 3))  if \_\_name\_\_ == "\_\_main\_\_":  unittest.main() |

Написать метод, возвращающий факториал числа:

|  |
| --- |
| import unittest  import math  def fact(n):  if n == 0 or n == 1:  return 1  else:  return n\*fact(n-1)  class TestFact(unittest.TestCase):  def test\_run(self):  for i in range(10):  self.assertEqual(fact(i), math.factorial(i))  if \_\_name\_\_ == "\_\_main\_\_":  unittest.main() |

Написать генератор, возвращающий факториалы чисел от 0 до n:

|  |
| --- |
| import unittest  import math  def fact(n):  f, i = 1, 0  while i < n:  yield f  i += 1  f = f\*i  class TestFact(unittest.TestCase):  def test\_run(self):  arr = tuple(fact(5))  true\_arr = tuple(math.factorial(i) for i in range(5))  self.assertEqual(arr, true\_arr)  if \_\_name\_\_ == "\_\_main\_\_":  unittest.main() |

**Менеджеры контекста** используются для управления ресурсами, а именно позволяют автоматизировать освобождение ресурсов. В python менеджер контекста реализан как объект с методами \_\_enter\_\_() и \_\_exit\_\_\_(). \_\_enter\_\_() выполняется при входе в блок with и возвращает ссылку на объект, которая присваивается переменной после ключевого слова as. \_\_exit\_\_\_() выполняется при выходе из контекста.

Написать менеджер контекста для файлов

|  |
| --- |
| class OpenFile:  def \_\_init\_\_(self, path, method):  self.f = open(path, method)  def \_\_enter\_\_(self):  return self.f  def \_\_exit\_\_(self, \*args):  self.f.close()  if \_\_name\_\_ == "\_\_main\_\_":  with OpenFile("test.txt", "w") as f:  f.write("hello") |

**Граф** – объект, состоящий из набора вершин и связывающих их рёбер.

**Невзвешенный граф** – граф, все ребра которого имеют одинаковый вес.

**Связный граф** – граф, между любой парой вершин которого существует хотя бы один путь.

**Компонента связности** – максимальный связный подграф графа.

**Обход графа в ширину** – метод обхода невзвешенного графа, заключающийся в том, что сначала посещаются все вершины, удалённые от заданной вершины на 1, потом все вершины, удалённые на 2 и так далее. В результате получаем величину удаления каждой вершины от заданной вершины, а также находим вершины, до которых нет пути (т.е. не входящие в текущую компоненту связности).

**Сложности алгоритма на списке смежностей.** Каждая вершина попадает в очередь и извлекается из очереди только один раз, поэтому время работы с очередью составляет O(V). У каждой вершины перебираются все её связи (рёбра), таким образом общее количество операций над рёбрами равно 2E, а время на работу с рёбрами O(E). Общее время работы алгоритма составляет O(V+E).

|  |
| --- |
| from collections import deque  def breadth\_first\_search(adjacencies, vertex):  level = [-1] \* len(adjacencies)  level[vertex] = 0  processing\_queue = deque()  processing\_queue.appendleft(vertex)  while processing\_queue:  v = processing\_queue.pop()  for adj in adjacencies[v]:  if level[adj] == -1:  processing\_queue.appendleft(adj)  level[adj] = level[v] + 1  return level  if \_\_name\_\_ == "\_\_main\_\_":  adjancencies = [  [1, 2, 3, 4], # 0  [0, 3, 5], # 1  [0, 4, 6], # 2  [0, 1, 4], # 3  [0, 2, 3], # 4  [1, 7], # 5  [2, 7], # 6  [5, 6] # 7  ]  level = breadth\_first\_search(adjancencies, 0)  print(level) |

**Обход графа в глубину** – метод обхода невзвешенного графа, состоящий в том, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

|  |
| --- |
| def depth\_first\_search(adjacencies, vertex, visited=None):  if not visited:  visited = [False] \* len(adjacencies)  visited[vertex] = True  for adj in adjacencies[vertex]:  if not visited[adj]:  depth\_first\_search(adjacencies, adj, visited)  return visited  if \_\_name\_\_ == "\_\_main\_\_":  adjancencies = [  [1, 2, 3, 4], # 0  [0, 3, 5], # 1  [0, 4, 6], # 2  [0, 1, 4], # 3  [0, 2, 3], # 4  [1, 7], # 5  [2, 7], # 6  [5, 6], # 7  [9], # 8  [8] # 9  ]  visited = depth\_first\_search(adjancencies, 0)  for v in range(len(adjancencies)):  if not visited[v]:  depth\_first\_search(adjancencies, v, visited)  print(visited) |