**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") |