MergeSort

- Напишите генератор, который принимает на вход два списка по возрастающим последовательностям и выдает одну объединенную возрастающую последовательность.
- Используя предыдущий генератор напишите еще один, который принимает на вход список и, используя сортировку слиянием, возвращает элементы в отсортированном порядке.
- Создайте собственный класс-итератор, который в качестве параметра конструктора принимает на вход число N и итерируется по N случайным числам.
- Отсортируйте этот итератор сортировкой слиянием.

Merge…

```
def generator(list1, list2):
		a = iter(list1)
		b = iter(list2)
		try:
			afirst = next(a)
			aempty = 0
		except StopIteration:
			aempty = 1
		try:
			bfirst = next(b)
			bempty = 0
		except StopIteration:
			bempty = 1
		while not aempty and not bempty:
			if afirst < bfirst:
				yield afirst
				try:
					afirst = next(a)
					aempty = 0
				except StopIteration:
					aempty = 1
			if bfirst <= afirst:
				yield bfirst
				try:
					bfirst = next(b)
					bempty = 0
				except StopIteration:
					bempty = 1
		if not aempty:
			yield afirst
			while 1:
				yield next(a)
		if not bempty:
			yield bfirst
			while 1:
				yield next(b)
```

```python
def generator(itA, itB):
        try:
                afirst = next(itA)
                aempty = 0
        except StopIteration:
                aempty = 1
        try:
                bfirst = next(itB)
                bempty = 0
        except StopIteration:
                bempty = 1
        while not aempty and not bempty:
                if afirst < bfirst:
                        yield afirst
                        try:
                                afirst = next(itA)
                                aempty = 0
                        except StopIteration:
                                aempty = 1
                if bfirst <= afirst:
                        yield bfirst
                        try:
                                bfirst = next(itB)
                                bempty = 0
                        except StopIteration:
                                bempty = 1
        if not aempty:
                yield afirst
                while 1:
                        yield next(itA)
        if not bempty:
                yield bfirst
                while 1:
                        yield next(itB)
```

```
def generator(itA, itB):
	try:
		afirst = next(itA)
		aempty = 0
	except StopIteration:
		aempty = 1
	try:
		bfirst = next(itB)
		bempty = 0
	except StopIteration:
		bempty = 1
	while not aempty and not bempty:
		if afirst < bfirst:
			yield afirst
			try:
				afirst = next(itA)
				aempty = 0
			except StopIteration:
				aempty = 1
		if bfirst <= afirst:
			yield bfirst
			try:
				bfirst = next(itB)
				bempty = 0
			except StopIteration:
				bempty = 1
	if not aempty:
		yield afirst
		yield from itA
	if not bempty:
		yield bfirst
		yield from itB
```

```
def generator(itA, itB):
        try:
                afirst = next(itA)
                aempty = 0
        except StopIteration:
                aempty = 1
        try:
                bfirst = next(itB)
                bempty = 0
        except StopIteration:
                bempty = 1
        while not aempty and not bempty:
                if afirst < bfirst:
                        yield afirst
                        try:
                                afirst = next(itA)
                                aempty = 0
                        except StopIteration:
                                aempty = 1
                if bfirst <= afirst:
                        yield bfirst
                        try:
                                bfirst = next(itB)
                                bempty = 0
                        except StopIteration:
                                bempty = 1
        yield afirst if not aempty else bfirst
        yield from itertools.chain(itA, itB)
```

```python
def generator(itA, itB):
        afirst = bfirst = None
        try:
                afirst = next(itA)
                bfirst = next(itB)
                while True:
                        if afirst < bfirst:
                                yield afirst
                                afirst = None
                                afirst = next(itA)
                        else:
                                yield bfirst
                                bfirst = None
                                bfirst = next(itB)
        except StopIteration:
                yield afirst if (bfirst == None) else bfirst
                yield from itertools.chain(itA, itB)
```

```python
def generator(itA, itB):
        afirst = bfirst = None
        try:
                afirst = next(itA)
                bfirst = next(itB)
                while True:
                        if afirst < bfirst:
                                afirst = yield afirst
                                afirst = next(itA)
                        else:
                                bfirst = yield bfirst
                                bfirst = next(itB)
        except StopIteration:
                yield afirst if (bfirst == None) else bfirst
                yield from itertools.chain(itA, itB)
```

```
def generator(itA, itB):
        yield from heapq.merge(itA, itB)
```

MergeSort…

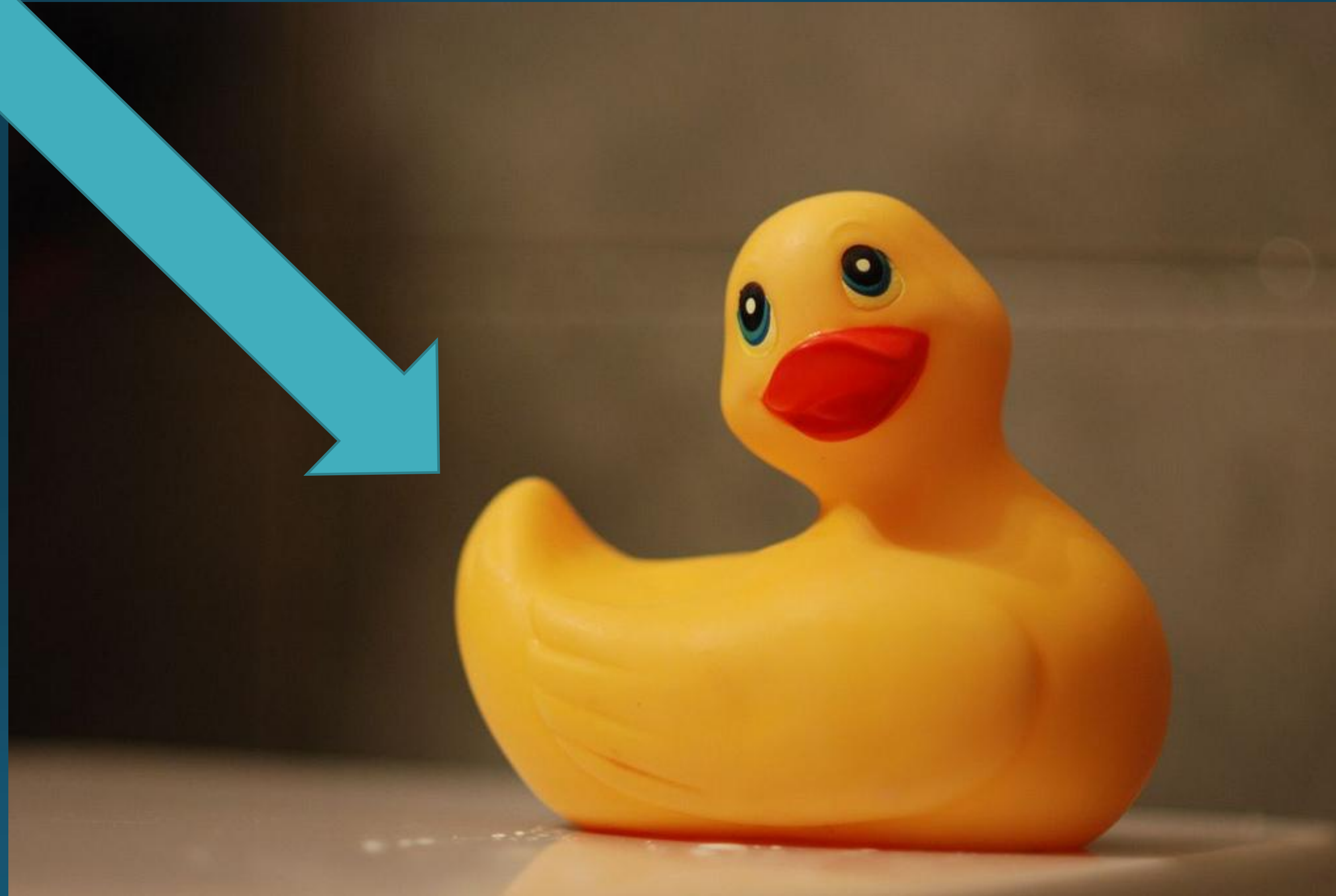
MergeSort...

```
def mergeSort(iterator):
	it = (iter([x]) for x in iterator)
	try:
		while True:
			it1 = next(it)
			it2 = next(it)
			it = itertools.chain(it, (generator(it1, it2), ))
	except:
		yield from it1
```

ООП

Python

Это уточка…

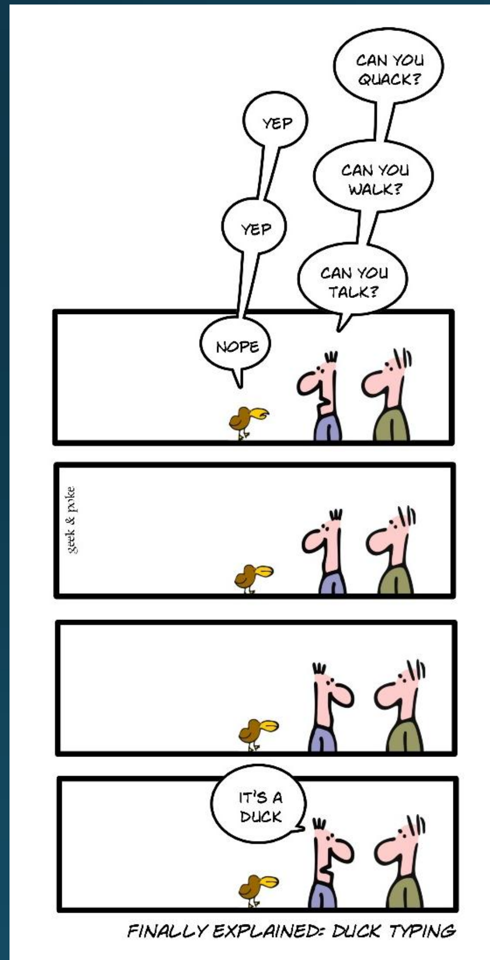


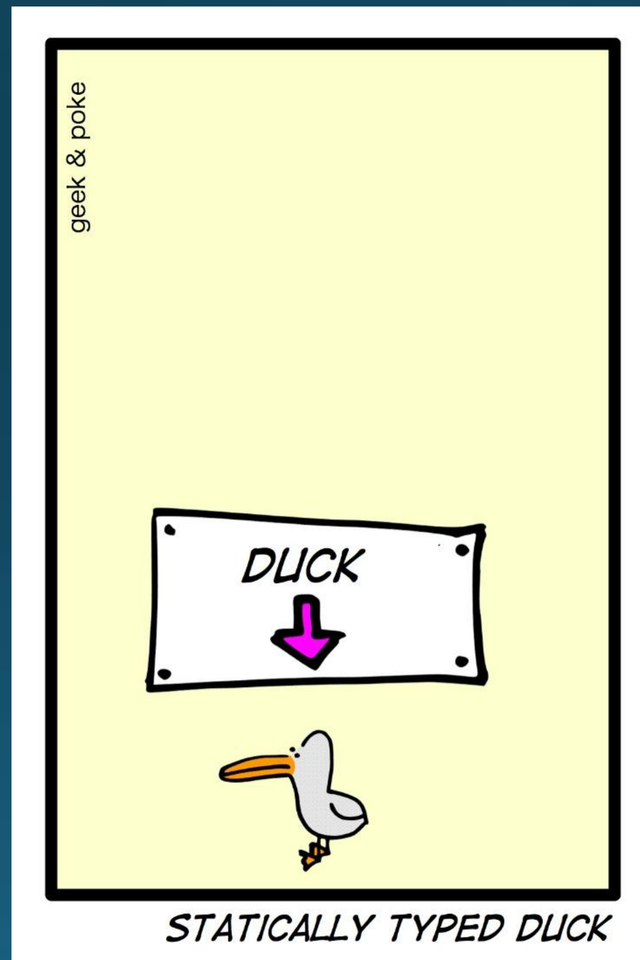
Утиная типизация

- «If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck».
- «Если оно выглядит как утка, плавает как утка и крякает как утка, то это, наверное, и есть утка».

- Утиная типизация - вид динамической типизации, когда границы использования объекта определяются его текущим набором методов и свойств, в противоположность наследованию от определённого класса

Утиная типизация





Принципы ООП

- Все данные представляются объектами
- Программа является набором взаимодействующих объектов, посылающих друг другу сообщения
- Каждый объект имеет собственную часть памяти и может иметь в составе другие объекты
- Каждый объект имеет тип
- Объекты одного типа могут принимать одни и те же сообщения (и выполнять одни и те же действия)
- Абстракция, инкапсуляция, полиморфизм, наследование

Определение класса

```
class имя_класса (надкласс1, надкласс2, ...):
	# определения атрибутов и методов класса
```

- У класса могут быть базовые (родительские) классы (надклассы), которые (если они есть) указываются в скобках после имени определяемого класса.

- Минимально возможное определение класса выглядит так:

```
class A:
	pass
```

Методы класса

Определение метода m1 класса A:

```
class A:
        def m1(self, x):
                # блок кода метода
```

Объявление поля (атрибута/свойства) класса A:

```
class A:
        attr1 = 2 * 2
```

В Питоне класс не является чем-то статическим после определения, поэтому добавить атрибуты можно и после объявления класса:

```
class A:
        pass
def myMethod(self, x):
        return x * x
A.m1 = myMethod
A.attr1 = 2 * 2
```

Создание экземпляра класса (инстанцирование)

```
class Point:
    def __init__(self, x, y, z):
        self.coord = (x, y, z)
    def __repr__(self):
        return "Point(%s, %s, %s)" % self.coord

p = Point(0.0, 1.0, 0.0)
p
>> Point(0.0, 1.0, 0.0)
```

Конструктор и «деструктор»

```
class Line:
        def __init__(self, p1, p2):
                self.line = (p1, p2)
        def __del__(self):
                print("Удаляется линия %s - %s" % self.line)

l = Line((0.0, 1.0), (0.0, 2.0))
del l
>> Удаляется линия (0.0, 1.0) - (0.0, 2.0)
```

- Это не деструктор – это «финализатор»
- в python реализовано автоматическое управление памятью (garbage collector);
- необработанные в «деструкторе» исключения игнорируются.

Инкапсуляция

Методы и данные объекта доступны через его атрибуты:

- _attribute # метод не предназначен для использования вне методов класса (или вне функций и классов модуля), однако, атрибут все-таки доступен по этому имени
- __attribute # атрибут перестает быть доступен по этому имени
- __init__ # атрибут доступен по своему имени, но его использование зарезервировано для специальных атрибутов, изменяющих поведение объекта

Инкапсуляция

```
>>> class A:
        _foo = 1
        __bar = 3
        __foobar__ = 4
>>> a = A()
>>> a._foo
1
>>> a.__foobar__
4
>>> a.__bar
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    a.__bar
AttributeError: 'A' object has no attribute '__bar'
```

Отличие __attr от C++ private

```
class parent(object):
      def __init__(self):
          self.__f = 2
      def get(self):return self.__f
class child(parent):
      def __init__(self):
        self.__f = 1
        parent.__init__(self)
      def cget(self):return self.__f

c = child()
c.get()
>> 2
c.cget()
>> 1
```

Почему так происходит?

```
>>>c._child__f
1
>>>c._parent__f
2
```

На самом деле у объекта "c" два разных атрибута!

Как нам скрыть что-то?

Или хотя бы запретить изменять?

Использовать:

- property
- __getatr__ и __setattr__
- __getatribute__

Property

```
class A:
    def __init__(self, x):
        self._x = x
    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "Свойство x“)
print(a.x)
a.x = 5
```

Property – другой вариант

```
class blah(object):
	@property # just a decorator
	def name(self):
		return self._x;
	@name.setter
	def name(self, value):
		self._x = value

>>>b = blah()
>>>b.name = 2
>>>b.name
2
```

Порядок разрешения доступа к методам и полям

Последовательность действий, производимых интерпретатором при разрешении запроса object.field (поиск прекращается после первого успешно завершённого шага, иначе происходит переход к следующему шагу).

- Если у object есть метод __getattribute__, то будет вызван он с параметром 'field' (либо __setattr__ или __delattr__ в зависимости от действия над атрибутом)
- Если у object есть поле __dict__, то ищется object.__dict__['field']
- Если у object.__class__ есть поле __slots__, то 'field' ищется в object.__class__.__slots__
- Проверяется object.__class__.__dict__['fields']
- Производится рекурсивный поиск по __dict__ всех родительских классов (при множественном наследовании поиск производится в режиме deep-first, в том порядке как базовые классы перечислены в определении класса-потомка).
- Если у object есть метод __getattr__, то вызывается он с параметром 'field'
- Возбуждается исключение AttributeError .

Полиморфизм

доступ на этапе исполнения => в python все методы виртуальные

```
class Parent:
        def isParOrPChild(self):
                return True
        def who(self):
                return 'parent'
class Child(Parent):
        def who(self):
                return 'child'
x = Parent()
x.who(), x.isParOrPChild()
>> ('parent', True)
x = Child()
x.who(), x.isParOrPChild()
>> ('child', True)
```

Полиморфизм

- Signature-oriented polymorfism
- Явно указав имя класса, можно обратиться к методу родителя (как впрочем и любого другого объекта).

```
class Child(Parent):
    def __init__(self):
        Parent.__init__(self)
```

- В общем случае для получения класса-предка применяется функция super.

```
class Child(Parent):
    def __init__(self):
        super(Child, self).__init__(self)
```

Имитация чисто виртуальных методов (NotImplementedException)

```
class abstobj(object):
    def abstmeth(self):
        raise NotImplementedError('Method abstobj.abstmeth is pure virtual')
abstobj().abstmeth()
>> Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in method
NotImplementedError: Method abstobj.abstmeth is pure virtual
```

Еще немного декораторов

```
class Spam(object):
    @classmethod
    def foo(cls, *args):  # classmethod takes a class as a first argument
        print("Foo " + cls.__name__)
    @staticmethod
    def bar(*args):
        print("Bar")

class Eggs(Spam):
    pass

s = Spam()
s.foo()    # Foo Spam
Spam.foo() # Foo Spam

e = Eggs()
e.foo()    # Foo Eggs
Eggs.foo() # Foo Eggs

Spam.bar() # Bar
e.bar()    # Bar
```

Управление иерархией наследования

Изменение атрибута __class__:

```
c = child()
c.val = 10
c.who()
>> ‘child‘
c.__class__ = parent
c.who()
>> ‘parent‘
c.val
>> 10
```

Имитация встроенных типов

```
>>> class Point:
...     def __init__(self, x = 0, y = 0):
...         self.x = x
...         self.y = y
...     def __add__(self, point):
...         return Point( self.x+point.x, self.y+point.y )
...     def __repr__(self):
...         return str(self.x) + " " + str(self.y);
>>> a = Point(1,2)
>>> b = Point(3,4)
>>> print(a+b)
4 6
```

Магические методы

```
__new__ __init__ __del__                      # init/final.
__repr__ __str__ __int__ __bool__             # conversions
__lt__ __gt__ __eq__ ...                      # comparisons
__add__ __sub__ __mul__ ...                   # arithmetic
__call__ __hash__ ...
__getattr__ __setattr__ __delattr__
__getitem__ __setitem__ __delitem__
__len__ __iter__ __contains__
```

Отношения между классами

Наследование и множественное наследование

```
class Par1(object):
        def name1(self): return 'Par1'
class Par2(object):
        def name2(self): return 'Par2'
class Child(Par1, Par2): # создадим класс,
                # наследующий Par1, Par2 (и,
                # опосредованно, object)
        pass
x = Child()
x.name1(), x.name2()
>> 'Par1','Par2'
```

«Строгая» типизация

```
class A:
    __slots__ = ('a', 'b')
    def __init__(self):
        self.a = 1
        self.b = 2

>>> a = A()
>>> a.a = 3
>>> a.a
3
>>> a.b
2
>>> a.v = 2
Traceback (most recent call last):
  File "<pyshell#108>", line 1, in <module>
    a.v = 2
AttributeError: 'A' object has no attribute 'v'
```

Модули

Python

Модуль

Модуль – файл, содержащий определения и другие инструкции языка Python. Имя файла образуется путем добавления к имени модуля суффикса(расширения) '.py'.

В пределах модуля его имя доступно глобальной переменной __name__.

Как написать модуль

Модуль оформляется в виде отдельного файла с исходным кодом.

```
#!/usr/bin/python # создали файл fibo.py
def fib(n):
    a, b = 0, 1
    while b < n:
        print(b)
        a, b = b, a+b
>>> import fibo
>>> fibo.fib(500) # 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Программа на Python

В программе на Python модуль представлен объектом-модулем, атрибутами которого являются имена, определенные в модуле:

```
#!/usr/bin/python
import datetime
d1 = datetime.date(2010, 03, 31)
```

В этом примере импортируется модуль datetime. В результате работы оператора import в текущем пространстве имен появляется объект с именем datetime.

Модули расширения

Модули для использования в программах на языке Python по своему происхождению делятся на обычные (написанные на Python) и модули расширения, написанные на другом языке программирования (как правило, на C).

С точки зрения пользователя они могут отличаться разве что быстродействием. Бывает, что в стандартной библиотеке есть два варианта модуля: **на Python и на C.** Таковы, например, модули **pickle и cPickle**.

Обычно модули на Python в чем-то гибче, чем модули расширения.

Подключение модуля

Подключение модуля к программе на Python осуществляется с помощью оператора import.

У него есть две формы: import и from-import:

- **import fibo**
- **from fibo import fib, fib2**
- **from fibo import ***
- import string as _string
- from anydbm import open as dbopen

С помощью первой формы с текущей областью видимости связывается только имя, ссылающееся на объект модуля, а при использовании второй - указанные имена (или все имена, если применена *) объектов модуля связываются с текущей областью видимости.