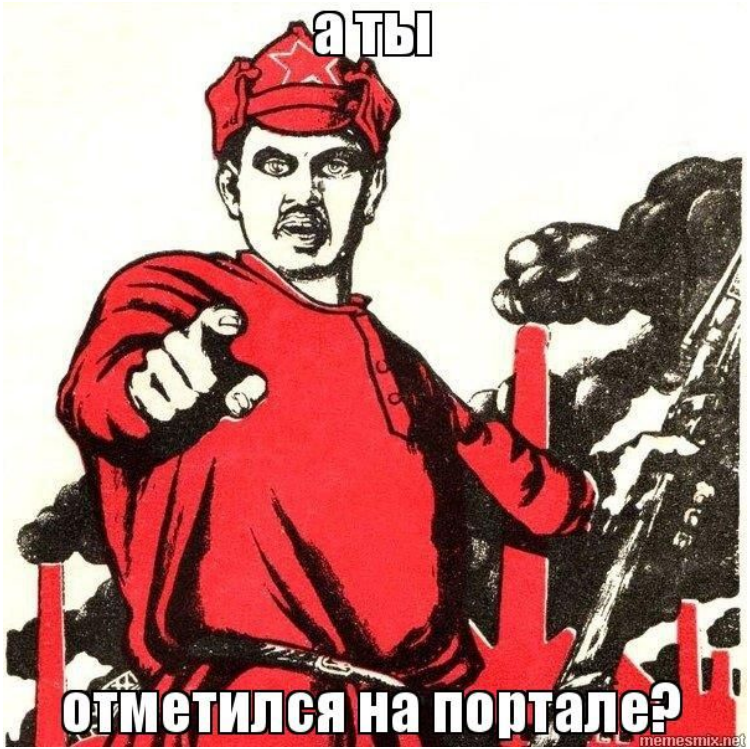
Python

Основы Python vol.2

Отметься на портале!



План занятия

1. О курсе
2. Структуры модулей
3. ООП, Классы в Python
4. Декораторы функция и классов
5. Исключения
6. Сортировки
7. Куча
8. Домашнее задание

Подготовимся к лекции

1. Переходим в директорию вашего форка
2. Открываем в ней терминал
3. git checkout master
4. git pull upstream master
5. git push -u origin master

Вспомним что было

- Python - интерпретируемый язык
- В котором все есть объект
- У нас есть dict, set, str, list, tuple, int, float, complex, bool, итд …
- пример передачи листов, интов, стринги в функции
- пример с дефолтными значениями например (.., a = [], ...)
- именованные аргументы. распаковка.
-

- теперь вы знаете почти все.

Заходим на kahoot.it

Поиграли, а что дальше?

- Бегло посмотрим на организацию python и его пакетов.
- Плавно от ООП перейдем к классам.
- Подробнее остановимся на том как их объявлять, узнаем что внутри у классов в python (
  атрибуты классов и экземпляра,
  классовые, статические, виртуальные, связанные, стандартные методы,
)
- Поговорим о дескрипторах, декораторах функций и классов.
- Узнаем о стандартных модулях functools, collections, itertools.
- Услышим про механизм обработки исключений в Python

и в самом конце

- Сортировки
- - Пузырьковая
- - Быстрая
- - Слиянием

- Куча
- Очередь с приоритетами

Питон и скрипты

- В Python принято оформлять свою программу в виде модулей - совокупности независимых “блоков” (а при образовании библиотечки - в виде пакетов)

- После импорта модуль представлен отдельным объектом, дающим доступ к пространству имён модуля.

```
import random
dir(random)
```

```
'_sha512',
'_sin',
'_sqrt',
'_test',
'_test_generator',
'_urandom',
'_warn',
'betavariate',
'choice',
'choices',
'expovariate',
'gammavariate',
'gauss',
'getrandbits',
'getstate',
'lognormvariate',
'normalvariate',
'paretovariate',
'randint',
```

ООП

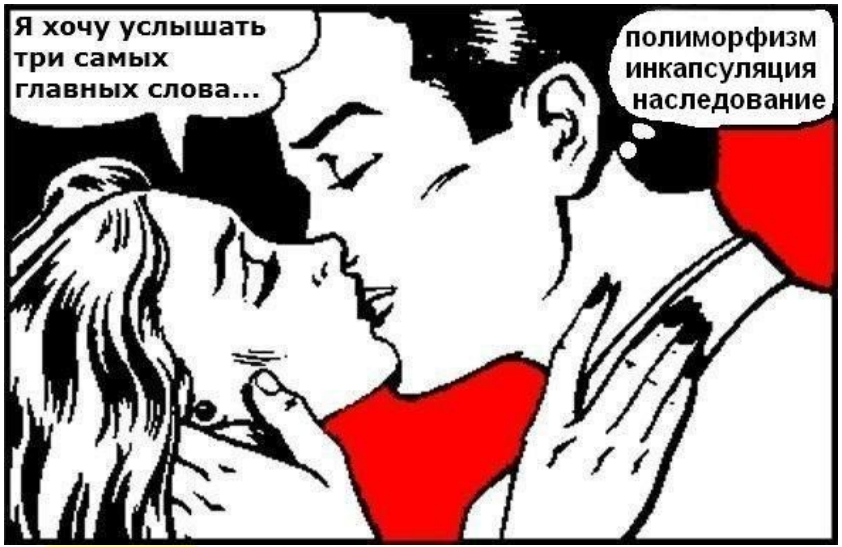
- что такое ооп?

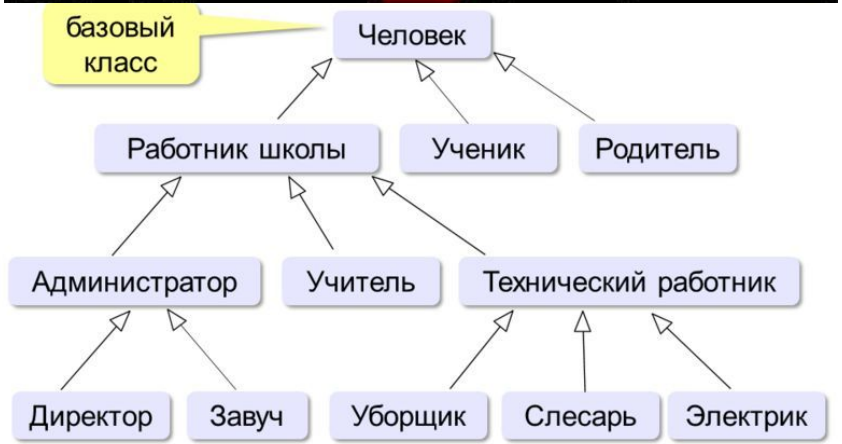
А что, если все объект?

ООП

- ООП - подход к программированию как к моделированию информационных объектов (wiki)
- Основное про ООП:
  - Все есть объект
  - Программа - совокупность объектов, которые взаимодействуют через запросы.
  - Объект имеет память
  - У каждого объекта есть тип

По сути отражаем реальную жизнь в коде,
Уменьшаем сложность восприятия





Например, хотим автоматизировать школу

Все есть объект

```
age = 35
print(age.__class__)
```

```
<class 'int'>
```

```
name = 'bob'
print(name.__class__)
```

```
<class 'str'>
```

```
def function(): pass
print(function.__class__)
```

```
<class 'function'>
```

```
class Bar(object): pass
b = Bar()
print(b.__class__)
```

```
<class '  main  .Bar'>
```

```
print('id for Bar\'s exemplar = {}'.format(id(b)))
```

```
id for Bar's exemplar = 139739489979080
```

```
print(Bar.__class__)
print('id for Bar = {}'.format(id(Bar)))
```

```
<class 'type'>
id for Bar = 15035848
```

id() возвращает идентификатор объекта (адрес объекта в памяти)

Помним, что *некоторые объекты могут иметь один и тот же идентификатор*, например: мелкие целые (с -5 по 256), True и False.

Классы в Python

Пример создания класса

__new__(cls, *ar, **kw)

__init__(self, *ar, **kw)

```
class MyFirstClass(object):
    """
    This is my first class.
    """
    classatribute = 99999

    def __init__(self, base=2):
        # self - указатель на экземпляр.
        self.base = base

    def strange_transform(self, x):
        return str(x) + ' (^_^) ' + str(self.base)
```

```
exemplar = MyFirstClass(3)
```

```
exemplar.strange_transform(3)
```

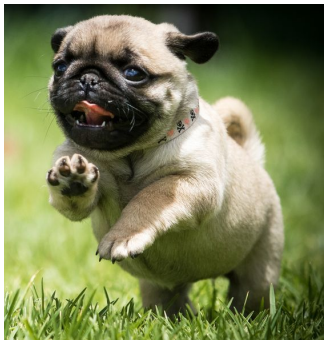
```
'3 (^_^) 3'
```

Модификаторы доступа

Концепция Python “мы все тут взрослые”:

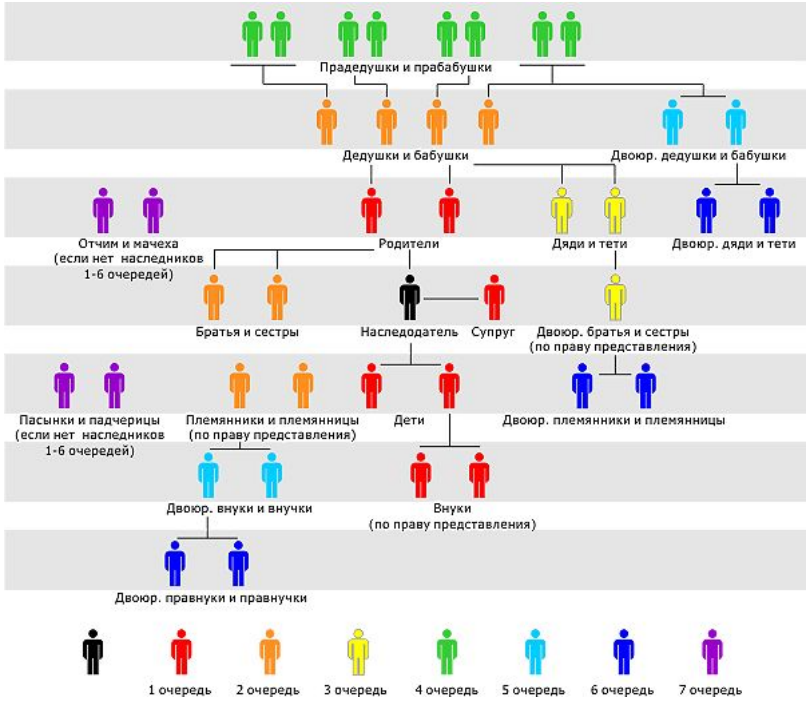
- нет никаких public/private/protected методов
- есть некоторые соглашения на то чтобы:
  - называть внутренние атрибуты с _something (все-равно легко достать)
  - называть то что совсем хочется скрыть с _ _something
    (можно достать если обратиться через ._ClassName_ _ something)
  -

В ноутбук!



Наследование

- Инструмент для повторного использования кода.
- Python поддерживает наследование классов, в т.ч. множественное.
- При множественном наследовании используется алгоритм линеаризации иерархии наследования - “C3”
  - дети вызываются раньше родителей
  - родители вызываются в порядке перечисления



https://www.python.org/download/releases/2.3/mro/

Наследование

super() -
вызывает следующий по иерархии наследования класс

help() - может показать иерархию

```
class A(object):

    def __init__(self):
        self.class_ = "A"
        self.batya = 'object'

    def come_on(self, python):
        print('A.{}'.format(python))

    def __repr__(self):
        return 'batya={}, class={}'.format(self.batya, self.class_)


class B(A):

    def __init__(self):
        super(B, self).__init__() # same as super().__init__()
        self.batya = self.class_
        self.class_ = "B"

    def come_on(self, python):
        print('B.{}'.format(python))

    def __repr__(self):
        return 'batya={}, class={}'.format(self.batya, self.class_)
```

```
a = A()
b = B()
```

```
print(a)
```

```
print(b)
```

Наследование

Проверки:

- isinstance(instance, CLASS)
  Проверяет что объект instance является
  экземпляром класса CLASS

- issubclass(CLASS_1, CLASS_2)
  Проверяет что CLASS_1 наследник CLASS_2

zarobuchek =

isinstance(zarobucher, Zarobuchki)
isinstance(zarobucher, Vorobuchek)
isinstance(zarobucher, Zayace)
issubclass(Zarobushki, Vorobuchek)
issubclass(Zarobushki, Zayace)

Состав классов и экземпляров

Что есть в классах можно узнать вызвав dir()

- Много внутренних методов обозначаемых как _ _метод_ _()
- атрибуты класса и экземпляра, методы

Где узнать, за что отвечает каждый метод?

```
dir(exemplar)
```

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'base',
 'classatribute',
 'strange_transform']
```

Состав классов и экземпляров

Что есть в классах можно узнать вызвав dir()

- Много внутренних методов обозначаемых как _ _метод_ _()
- атрибуты класса и экземпляра, методы

Где узнать, за что отвечает каждый метод?

- догадаться из опыта и названия
- вызвать help
- в гугле
- на следующих слайдах будут ссылки ^_^ не на гугл конечно

```
dir(exemplar)
```

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'base',
 'classatribute',
 'strange_transform']
```

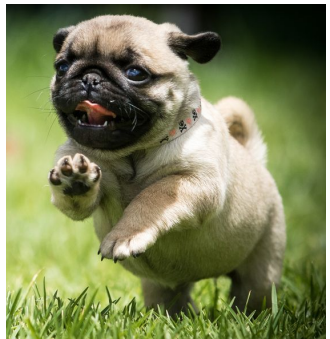
__dict__ в классах

Все атрибуты объекта доступны в виде словаря _ _ dict _ _

Добраться до словаря можно используя функцию vars() или напрямую

- По словарю делается доступ к атрибутам (__dict__ объекта, затем __dict__ класса) и методам (__dict__ класса))
- Его можно изменять “на лету”
- В противопоставление ему есть механизм __slots__, который позволяет зафиксировать набор атрибутов.

В ноутбук



Методы с __Underline__

Переопределяем операторы

- __add__() для “+”
- __sub__() для “-”
- __contains__() для “in”
- __eq__() для “==”
- __lt__() для “<”

```
class Sesese():

    def __init__(self, a):
        self.atr = a

    def __add__(self, something):
        if isinstance(something, int):
            return self.atr + something
        elif isinstance(something, Sesese):
            return self.atr + something.atr
        else:
            raise Exception('I know python!')
```

```
obj = Sesese(3)
```

```
obj + 43
```

```
46
```

```
obj + Sesese(15), obj.__add__(Sesese(15))
```

```
(18, 18)
```

```
obj + 'sd'
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-494-17865239dfde> in <module>()
----> 1 obj + 'sd'

<ipython-input-487-5f2a03f1c25e> in __add__(self, something)
     10             return self.atr + something.atr
     11         else:
---> 12             raise Exception('I know python!')

Exception: I know python!
```

Методы с __Underline__

Переопределяем операторы

- __getitem__() для “[]”

- и еще куча методов
- и т.д. и т.п.

Список методов

https://docs.python.org/3/reference/datamodel.html?#emulating-numeric-types
http://www.siafoo.net/article/57
https://pythonworld.ru/osnovy/peregruzka-operatorov.html
https://www.python-course.eu/python3_magic_methods.php

```
class Sesese():

    def __init__(self, some_list_example):
        self.mem = some_list_example

    def __getitem__(self, index):
        print('item is {}'.format(index))
        if isinstance(index, int):
            return self.mem[index % len(self.mem)]
        elif isinstance(index, slice):
            return self.mem[index]
        else:
            raise Exception('Type Err! I know Python!')
```

```
a = Sesese([1,2,3,4,5,6,7,8])
```

```
a[4]
```

```
item is 4
5
```

```
a[1:6:2]
```

```
item is slice(1, 6, 2)
[2, 4, 6]
```

```
a['str']
```

```
item is str
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-481-bc1124e219fc> in <module>()
----> 1 a['str']

<ipython-input-477-89f7f3025de8> in __getitem__(self, index)
     11             return self.mem[index]
     12         else:
---> 13             raise Exception('Type Err! I know Python!')

Exception: Type Err! I know Python!
```

Методы с __Underline__

Переопределяем операторы

- __enter__() и __exit__()
  для контекстного менеджера

!ДЗ на доп баллы (3) описать, что такое контекстный менеджер в 1 слайде

С мемасиками конечно же!

Методы с __Underline__

Написать свое комплексное число.

- Имя класса: Complex_num.
- может принимать ТОЛЬКО 2 аргумента - мнимая и действительная часть (slots?)
- переопределить методы: __eq__, __str__, __add__, __hash__, __gt__

3 минуты, Вопросы?

Методы и что мы с ними можем сделать

- В Python только один метод с заданным именем, перегрузки явной нету.
- Если хочется, можно воспользоваться args/kwargs атрибутами функций.

```
class B:
    def __init__(self, a=34):
        self.var = a

    def bobo(self, integer, string):
        return str(integer) + string

    def bobo(self, string):
        return string
```

```
class Nopepe:

    def function(self, *args, **kwargs):
        print (args[3])
        if 'coco' in kwargs:
            print (kwargs['coco'])
```

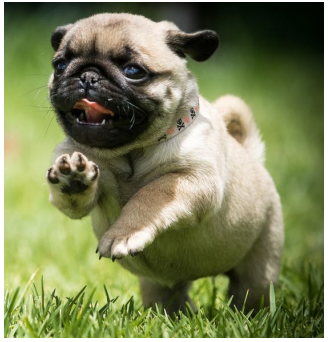
```
tmp_1 = Nopepe()
tmp_1.function(1, 2, 3, 4, 5, 6, j=84, coco='im_here', excel='wtf')
```

```
4
im_here
```

Декораторы классов и функций

Способ поменять поведение функции (класса) не изменяя ее код.
(функция, которой на вход подается объект и которая возвращает функцию)

- кеширующий декоратор
- декоратор выводящий время исполнения
- логирующий декоратор
- декоратор для проверки аргументов



декоратор (над классом) синглтон (?)

В ноутбук!

@classmethod и @staticmethod

```
class A(object):

    count_ = 100500

    def foo(self, x):
        print ('foo with self=%s and x=%s' % (self, x))

    @classmethod
    def class_foo(cls, x):
        print ('class foo with cls=%s and local_variables=%s \
including count_=%s' % (cls, dir(), cls.count_))

    @staticmethod
    def static_foo(x):
        print ('static foo x=%s and local_variables=%s' % (x, dir()))

a=A()
```

@classmethod и @staticmethod

- обычный метод принимает в качестве первого аргумента метода экземпляр (self)

- классовый метод принимает в качестве первого аргумента метода - класс экземпляра (cls), можно вызывать и из класса, и из экземпляра. Один из вариантов использования - наследуемые конструкторы.

- в статический метод не передаются в качестве первого аргумента ни экземпляр ни класс. похож на обычные функции, но можно вызывать из экземпляра или класса. нет доступа к атрибутам класса

Дескрипторы

В простонародье - свойства

В Python объект, у которого есть хотя бы один из методов __get__, __set__,__delete__

Когда мы хотим переопределить get/set на свой манер

https://compscicenter.ru/courses/python/2015-autumn/classes/1559/ - глубже о дескрипторах

```
class SomeBank:

    def __init__(self, start):
        self._rubles = start

    @property
    def rubles(self):
        return self._rubles

    @rubles.setter
    def rubles(self, value):
        if value > 0:
            self._rubles = value
        else:
            raise Exception('I know python!')
```

```
interv = SomeBank(6000)
```

```
interv.rubles
```

```
6000
```

```
interv.rubles = 2342
```

```
interv.rubles
```

```
2342
```

```
interv.rubles = -32
```

```
---------------------------------------------------------------
Exception                                 Traceback (
```

Дескрипторы

В простонародье - свойства

Когда мы хотим переопределить get/set на свой манер

```
# будьте внимательнее - где тут ошибка?

class MoneySaver:

    def __init__(self, exchange_rate):
        self.exchange_rate = exchange_rate
        self.copilka = 0

    @property
    def copilka(self):
        return self.copilka * self.exchange_rate

    @copilka.setter
    def copilka(self, x):
        self.copilka += x / self.exchange_rate

dollars_savers = MoneySaver(100)
dollars_savers.copilka = 10
print(dollars_savers.copilka)
```

Исключения

```
try:
    # Здесь код, который может вызвать исключение
    raise Exception("message")  # Exception, это один из стандартных типов исключения (всего лишь класс),
                                # может использоваться любой другой, в том числе свой
except (Тип исключения1, Тип исключения2, …) as Переменная:
    # Код в блоке выполняется, если тип исключения совпадает с одним из типов
    # (Тип исключения1, Тип исключения2, …) или является наследником одного
    # из этих типов.
    # Полученное исключение доступно в необязательной Переменной.
except (Тип исключения3, Тип исключения4, …) as Переменная:
    # Количество блоков except не ограничено
    raise  # Сгенерировать исключение "поверх" полученного; без параметров - повторно сгенерировать полученное
except:
    # Будет выполнено при любом исключении, не обработанном типизированными блоками except
else:
    # Код блока выполняется, если не было поймано исключений.
finally:
    # Будет исполнено в любом случае, возможно после соответствующего
    # блока except или else
```

Исключения

```
def f():
    try:
        print(1)
        try:
            print(4)
        finally:
            print(5)
            return 5
    finally:
        print (2)
        return 2

a = f()
a
```

executed in 11ms, finished 11:58:46 2018-07-12

```
def f():
    try:
        print(1)
        try:
            print(4)
        finally:
            print(5)
            #return 5
    finally:
        print (2)
        return 2

a = f()
a
```

executed in 10ms, finished 11:58:46 2018-07-12

Исключения

```
def f():
    try:
        print(1)
        try:
            print(4)
        finally:
            print(5)
            return 5
    finally:
        print (2)
        return 2

a = f()
a
```

executed in 11ms, finished 11:58:46 2018-07-12

```
1
4
5
2

2
```

```
def f():
    try:
        print(1)
        try:
            print(4)
        finally:
            print(5)
            #return 5
    finally:
        print (2)
        return 2

a = f()
a
```

executed in 10ms, finished 11:58:46 2018-07-12

```
1
4
5
2

2
```

Сортировки

Задача:

Дано N элементов на которых определено отношение порядка.

Нужно упорядочить эти N элементов в порядке неубывания т.е. найти такую перестановку записей после которой значения будут отсортированы.

Сортировки бывают:
- устойчивыми - когда элементы с одинаковыми значениями не меняют позиции после сортировки
- внутренними - в оперативной памяти
- внешними - во внешней памяти - например разных машинах

Сортировки. Пузырьковая

1. Производим повторяющийся проход по массиву N-1 раз (или пока
2. Сравниваем каждый элемент с соседним(правым) и меняем их местами если он больше
3. Повторяем это для пока не сравним этот элемент со всеми
4. Повторить сначала, пока не дойдем до конца

Сортировки. Быстрая

1. Из элементов выбирается опорный (чем ближе к медиане, тем лучше)
2. Массив разбивается на два:
   левая часть - элементы не больше опорного,
   правая часть - элементы не меньше опорного
3. Рекурсивно вызываются шаги 1 и 2 для левой и правой части

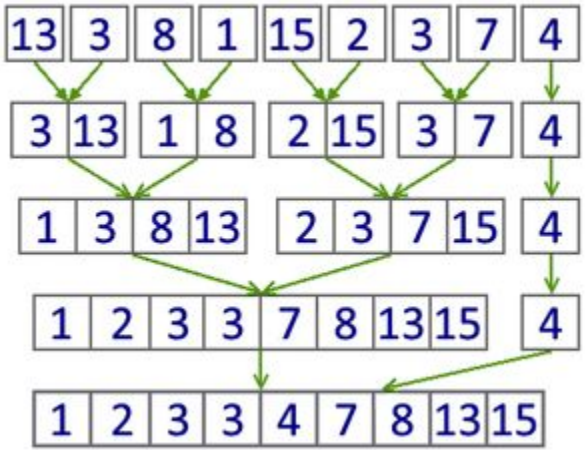
Среднее время O(n log n), худшее время O(n^2)

[Гифка о том, как работает]

Сортировки. Слиянием

1. Разбиваем массив на два примерно одинаковых
2. Для каждого подмассива вызывается процедура сортировки
3. Получившиеся отсортированные подмассивы объединяются в один

Среднее, худшее и лучшее время работы O (n log n) + доп. память O(n)



http://interactivepython.org/courselib/static/pythonds/SortSearch/TheMergeSort.html

Куча (max / min - heap)

Дерево в котором выполняется свойство: если узел B родитель узла A, то ключ B должен быть больше чем ключ A

Важное свойство: наибольший/наименьший элемент - всегда корень кучи

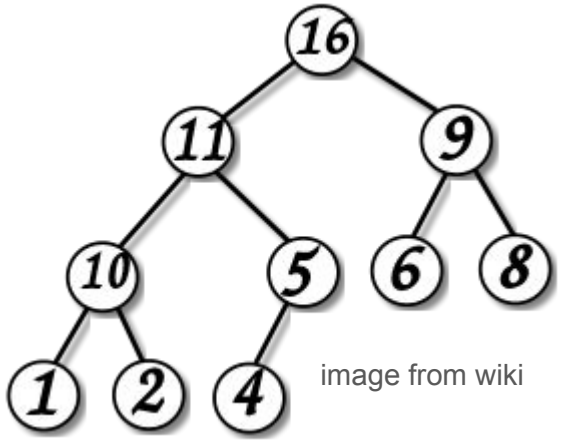


image from wiki

Двоичная Куча (max / min - heap)

Дерево в котором выполняются свойства:

- значение в любой вершине не меньше чем значения ее потомков
- глубина листьев отличается не более чем на 1 слой
- последний слой заполняется слева направо

- найти максимум (минимум) O(1)
- удалить максимум (минимум) O(log n)
- обновить ключ O(log n)
- добавить элемент O(log n)
- слияние куч O(n+m)
- построение кучи O(n)

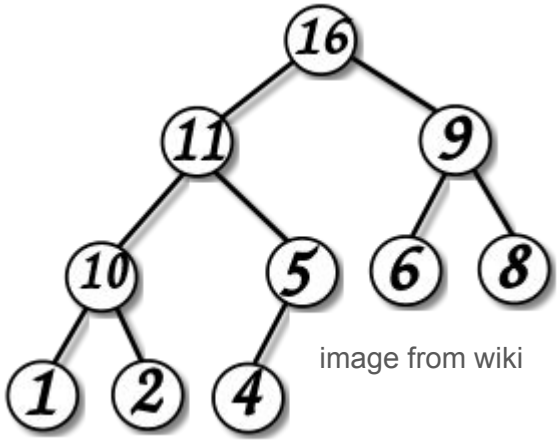
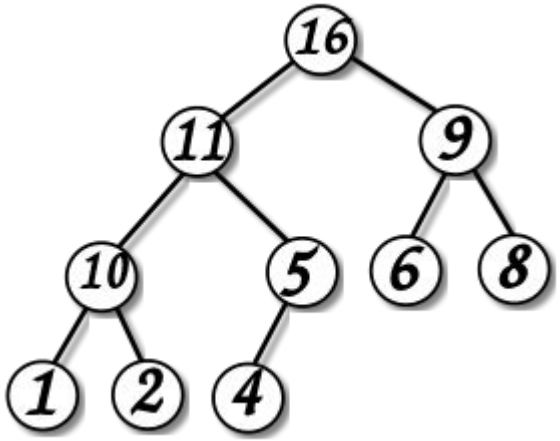


image from wiki

Двоичная Куча (max / min - heap)



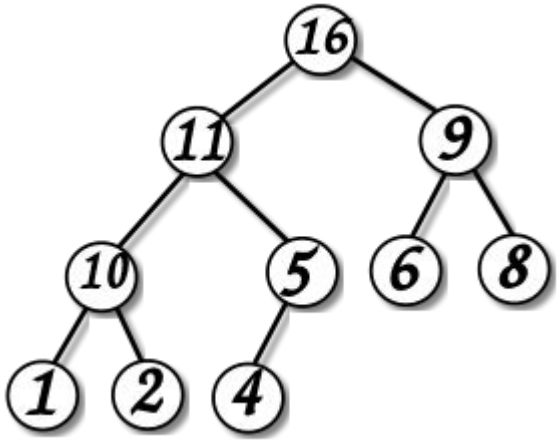
i - родитель

2 * i + 1 - левый потомок

2 * i + 2 - правый потомок

[16, 11, 9, 10, 5, 6, 8, 1, 2 ,4]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Куча. Хранение



i - родитель

2 * i + 1 - левый потомок

2 * i + 2 - правый потомок

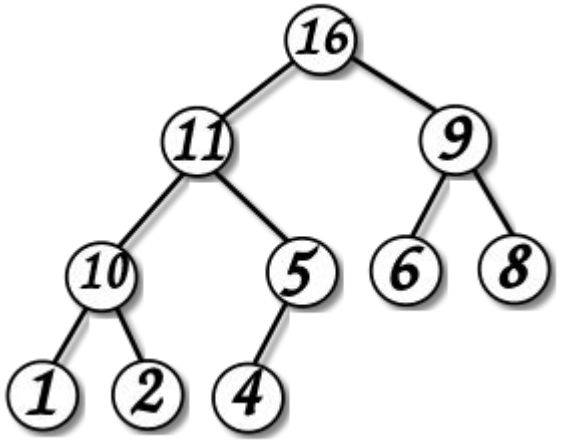
[16, 11, 9, 10, 5, 6, 8, 1, 2 ,4] - такое дерево можно хранить в массиве
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Это экономно.

Куча. Процедуры

- восстановление свойств

```
def _shift_down(array, i):
    # помогаем "утопить" мелкий элемент
    left = 2 * i + 1
    right = 2 * i + 2
    largest = i

    heap_size = len(array) - 1
    if left <= heap_size and array[left] > array[largest]:
        largest = left
    if right <= heap_size and array[right] > array[largest]:
        largest = right

    if largest != i:
        array[i], array[largest] = array[largest], array[i]
        _heapify(array, largest)
        # _heapify для потомка с новым значением
        # получается опускаем вниз элемент-родитель
        # если он меньше потомка
```

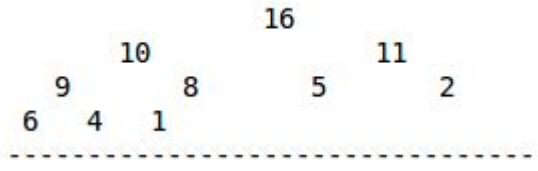


Куча. Процедуры

- создание кучи из неупорядоченного массива

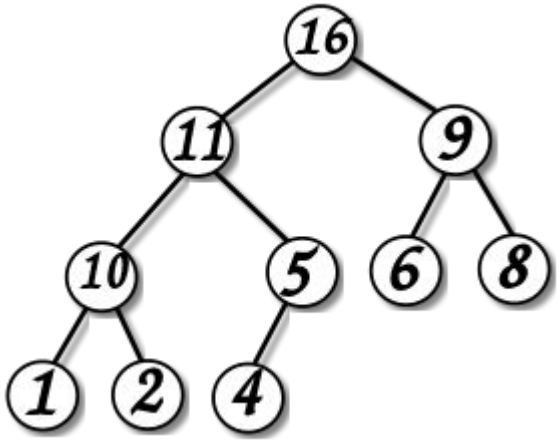
```
def build_heap(array):
    heap_size = len(array) - 1
    for i in range(0, heap_size // 2 + 1)[::-1]:

        _shift_down(array, i)
        print(array, i)
```

```
[0,  1,  2,  3,  4,  5,  6,  7,  8,  9]
[11, 9,  5,  6,  1, 16,  2, 10, 4,  8]
[11, 9,  5,  6,  8, 16,  2, 10, 4,  1] 4
[11, 9,  5,  10, 8, 16,  2,  6,  4,  1] 3
[11, 9,  16, 10, 8,  5,  2,  6,  4,  1] 2
[11, 10, 16, 9,  8,  5,  2,  6,  4,  1] 1
[16, 10, 11, 9,  8,  5,  2,  6,  4,  1] 0
```



Куча. Процедуры

- Вытащить максимум (+ сохранить свойство кучи)
  - Взять нулевой элемент массива
  - на его место поставить последний элемент
  - запустить для него процедуру shift_down для корня



Куча. Процедуры

- Добавить новый элемент с приоритетом (+ сохранить свойство кучи)
  - Добавить его в конец массива
  - запустить для него процедуру shift_up для корня

```
def _shift_up(array, i):
    # помогаем всплыть элементу
    parent = (i - 1) // 2
    while i > 1 and array[parent] < array[i]:

        array[parent], array[i] = array[i], array[parent]

        i = parent
        parent = (i - 1) // 2
```

Куча. Процедуры

- Изменить приоритет (+ сохранить свойство кучи)
  - Запомнить старый
  - заменить приоритет
  - запустить для него процедуру shift_up если новый приоритет больше, или shift_down если меньше

```
def change_priority(array, pos, new_prior):
    old_p = array[pos]
    array[pos] = new_prior
    if old_p > new_prior:
        _shift_down(array, pos)
    else:
        _shift_up(array, pos)
```

Удаление (сделать приоритет -inf, запустить shift_down, удалить из массива)

Очередь с приоритетами

Очередь - тип данных с операциями “добавить в конец”, “достать с начала”.

Очередь с приоритетами - “добавить элемент”, “достать максимальный”.

MUST READ

Источники

1. Про дескрипторы, понятно и просто
2. Довольно муторно, но с основ про классы
3. Анимашки про сортировки
4. Декораторы для новичков
5. Крутой курс про Python
6. Серия статей про ООП в Python
7. Если интересно : Про классы и метаклассы в питоне

Оставьте обратную связь!!!