

Основы программирования на Python3

Функции
Исключения
ООП

Разбор домашнего задания

- ▶ В целом – все молодцы! Уделяйте больше внимания запуску и проверке своих решений.
- ▶ Старайтесь не называть переменные встроенными типами данных: list, int, set, dict, float и т.д

```
list = [1, 2, 3, 4, 5]
```

```
for l in list:
```

```
...
```

```
list((1, 2, 3))
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'list' object is not callable

Разбор домашнего задания

- ▶ Будьте аккуратны со сравнением типа float

```
>>> 0.1 + 0.2
```

```
0.30000000000000004
```

```
>>> 0.1 + 0.2 == 0.3
```

```
False
```

Для правильной работы можно использовать тип Decimal

```
from decimal import Decimal
```



Разбор домашнего задания

- ▶ Сломанные копья – задача #6
- ▶ Все решения по своему хороши
- ▶ Встроенная функция zip:

```
a = '1sferg'
k = a[1::2]
v = a[::2]
tuple(zip(k, v))
dict(zip(k, v))
>>> (('s', '1'), ('e', 'f'), ('g', 'r'))
>>> {'s': '1', 'g': 'r', 'e': 'f'}
```

Разбор домашнего задания

- ▶ Как не тащить в репозиторий мусор в виде `.idea`, `.DS_store` и прочего?
- ▶ Пользуйтесь `.gitignore`!
- ▶ Папка: `lib/`
- ▶ Файл: `.idea`
- ▶ Маска: `*.egg`
- ▶ Если файл один раз закоммичен в GIT – он останется там навсегда, даже если вы его удалите
- ▶ Отсюда правило: никогда не хранить пароли в GIT

Зачем нужны функции?

```
a = input('Enter string')  
b = int(a)  
b += 0  
print(b)  
A = input('Enter string')  
b = int(a)  
b += 10  
print(b)
```


```
def inp_add():  
    a = input('Enter string')  
    b = int(a)  
    b += 10  
    print(b)
```

```
for i in range(2):  
    inp_add()
```

Жизнь без функций

Жизнь с функциями

Зачем нужны функции?

- ▶ DRY (Don't repeat yourself),
переиспользование кода
 - ▶ Обратная сторона DRY – WET (Write everything twice)
 - ▶ Разделение программы на логические сущности
 - ▶ Использование рекурсии
 - ▶ Наследование / инкапсуляция / полиморфизм
- 

Синтаксис

▶ `def func1():`
 `pass`

`def func2(a, b):`
 `pass`

`def func3(a, b, c=None):`
 `pass`

`def func4(a, *args):`
 `pass`

`def func5(*args, **kwargs):`
 `pass`

`def func6(a, b=1, *args, **kwargs):`
 `"""Some great description here"""`

Что это за звездочки?

- ▶ Распаковка аргументов

```
a, b, c = (1, 2, 3)
```

```
print(a, b, c)
```

```
>>> 1 2 3
```

```
a, *b = (1, 2, 3)
```

```
print(a, b)
```

```
>>> 1 [2, 3]
```

```
a, *b, c, d = (1, 2, 3, 5, 6)
```

```
print(a, b, c)
```

```
>>> 1 [2, 3] 5 6
```

Что за двойные звездочки?

- ▶ Только для передачи именованных параметров в функцию. Ключи должны быть сроками!

```
def kwargs_f(**kwargs):  
    print(kwargs)
```

```
kwargs_f(**{'a': '2', 'b': 3})
```

```
>>> {'a': '2', 'b': 3}
```

Когда использовать * и **?

- ▶ Количество аргументов неизвестно
- ▶ Названия аргументов неизвестны
- ▶ Быстрая конкатенация словарей:

```
a = {'a': 2}
```

```
b = {'b': 4}
```

```
dict(**a, **b)
```

```
>>> {'a': '2', 'b': 4}
```

__docstring__

- ▶ Взять за правило: написал функцию – описал докстринг
- ▶ Не стоит гнаться за описанием всех параметров – при быстрой разработке тяжело поддерживать актуальное описание

```
▶ def func6(a, b=None, *args, **kwargs):  
    """
```

```
    Nightmarish func with lot's of argzzzzz
```

```
    :param a:
```

```
    :param b:
```

```
    :param args:
```

```
    :param kwargs:
```

```
    :return:
```

```
    """
```

Функции – тоже объекты

```
def a(x):  
    x += 5  
    return x
```

```
def b(some_arg):  
    print(some_arg(1))
```

```
b(a)
```

```
>>> 6
```



Рекурсия

```
▶ def factorial(n):  
    if n < 1:  
        return 1  
    else:  
        return n * factorial(n - 1)
```

factorial(5)

>>> 6

Не забывайте писать условие выхода из рекурсии

Исключения – зачем?

- ▶ Защита программы от непредвиденного поведения в рантайме
- ▶ Безусловная передача информации об ошибке

Синтаксис

► **try:**

```
a = int('Это точно не инт')
```

except ValueError **as** e:

```
    print('Что-то пошло не так! %s' % e)
```

else:

```
    print('Все пошло как надо'))
```

finally:

```
    print('В конце я всегда уведомляю хозяина')
```


Синтаксис

- ▶ Обязателен хотя бы один блок `except` или `finally`
- ▶ Можно отлавливать сразу несколько типов ошибок:

- ▶ **try:**

```
    a = 1 / 0
```

```
except (ZeroDivisionError, TypeError):
```

```
    print('Нехорошо делить на 0!')
```

Безусловная передача ошибки

```
▶ a = int(input())
```

```
if a < 0:
```


```
    raise Exception('Недопустимы отрицательные  
числа')
```

- ▶ Ошибка будет передаваться по стеку вызовов функций, пока не будет отловлена или программа не завершится аварийно
- ▶ Соблазнительно для передачи внутренних ошибок, но следует использовать аккуратно - вызов дорогой

Широкий диапазон ошибок – плохо

- ▶ Все ошибки в Python наследуются от базового класса Exception
- ▶ Как следствие, эксепт Exception позволит поймать любую ошибку
- ▶ Например, незапланированный RuntimeError или KeyboardInterrupt

Какие ошибки ловить?

- ▶ `TypeError`: ошибка приведения типов
 - ▶ `ValueError`: ошибка работы со значением
 - ▶ `ZeroDivisionError`: деление на 0
 - ▶ `KeyError`: отсутствие ключа в словаре
 - ▶ `IndexError`: отсутствие индекса в массиве
 - ▶ `AttributeError`: отсутствие метода или атрибута данных у класса
 - ▶ `ImportError`: ошибка при импорте (не установлен пакет и др)
 - ▶ `KeyboardInterrupt`: нажатие ctrl-C на клавиатуре
- 

EAFP & LBYL

- ▶ Easier to Ask Forgiveness than Permission

```
d = {}
```

```
try:
```

```
    print(d['1'])
```

```
except KeyError:
```

```
    pass
```

- ▶ Look Before You Leap

```
if '1' in d:
```

```
    print(d['1'])
```

- ▶ Вариант на каждый день:

```
print(d.get('1', ''))
```

Три кита ООП

- ▶ Наследование (*inheritance*)


Способность наследовать методы и данные существующего типа

Растение [время жизни] –> Овощ [время жизни, название] –> Овощ Баклажан (экземпляр Овоща)

Три кита ООП

- ▶ Инкапсуляция (*encapsulation*)

Возможность определять видимость методов и данных внутри программы.

- ▶ Нет большой необходимости использовать в Python
 - ▶ Реализовано в Python не нативно
- 

Три кита ООП

- ▶ Полиморфизм (*polymorphie*)
- ▶ Функция способна обрабатывать разные типы данных
- ▶ Работает "из коробки" в языках с динамической типизацией