Лекция №3

Основы программирования на Python (продолжение)

- Контейнерные типы (коллекции): кортежи, списки, множества
- Словари
- Работа с файлами
- Модули и импорты
- Области видимости
- Рекурсия
- Декораторы
- Практика

Список (List) - Mutable

В чистом Python нет массивов с произвольным типом элемента. Вместо них используются списки. Их можно задать с помощью литералов, записываемых в квадратных скобках, или посредством списковых включений. Варианты задания списка приведены ниже:

```
lst1 = [1, 2, 3,]
lst2 = [x**2 for x in range(10) if x % 2 == 1]
lst3 = list("abcde")
```

Список как пример последовательности

Ниже обобщены основные методы последовательностей и функций для работы с последовательностями. Следует напомнить, что последовательности бывают неизменяемыми (immutable) и изменяемыми (mutable). Сначала функции для работы с последовательностями:

Функция	Описание
len(s)	Длина последовательности s.
x in s	Проверка принадлежности элемента последовательности. В новых версиях Python можно проверять принадлежность подстроки строке. Возвращает True или False.
x not in s	= not x in s
s + s1	Конкатенация последовательностей
s*n или n*s	Последовательность из n раз повторенной s. Если n < 0, возвращается пустая последовательность.
s[i]	Возвращает i-й элемент s или len(s)-i-й, если i < 0
s[i:j:d]	Срез из последовательности s от i до j с шагом d. Так же как и для строк.
min(s)	Наименьший элемент s
max(s)	Наибольший элемент s
s[i] = x	x i -й элемент списка s заменяется на x
s[i:j:d] = t	Срез от i до j (с шагом d) заменяется на (список) t
del s[i:j:d]	Срез от i до j (с шагом d) заменяется на (список) t

Список как пример последовательности

У последовательностей имеются также методы. Ниже представлены методы изменяемых последовательностей.

Метод	Описание
.append(x)	Добавляет элемент в конец последовательности.
.count(x)	Считает количество элементов, равных x.
.extend(s)	Добавляет к концу последовательности последовательность s.
.index(x)	Возвращает наименьшее i, такое, что s[i] == x. Выбрасывает исключение ValueError, если x не найден в s.
.insert(i, x)	Вставляет элемент x в i -й промежуток.
.pop(i)	Возвращает i -й элемент, удаляя его из последовательности.
.reverse(s)	Меняет порядок элементов s на обратный.
.sort([cmpfunc])	Сортирует элементы s. Может быть указана своя функция сравнения cmpfunc.

Взятие элемента по индексу и срезы

Здесь же следует сказать несколько слов об индексировании последовательностей и выделении подстрок (и вообще - подпоследовательностей) по индексам. Для получения отдельного элемента последовательности используются квадратные скобки, в которых стоит выражение, дающее индекс. Индексы последовательностей в Python начинаются с нуля. Отрицательные индексы служат для отсчета элементов с конца последовательности (-1 - последний элемент). Пример:

```
>>> s = [0, 1, 2, 3, 4]
>>> print s[0], s[-1], s[3]
0 4 3
>>> s[2] = -2
>>> print s
[0, 1, -2, 3, 4]
>>> del s[2]
>>> print s
[0, 1, 3, 4]
```

Взятие элемента по индексу и срезы

Удалять элементы можно только из изменчивых последовательностей и желательно не делать этого внутри цикла по последовательности.

Несколько интереснее обстоят дела со срезами. Дело в том, что в Python при взятии среза последовательности принято нумеровать не элементы, а промежутки между ними. Поначалу это кажется необычным, тем не менее, очень удобно для указания произвольных срезов. Перед нулевым (по индексу) элементом последовательности промежуток имеет номер 0, после него - 1 и т.д.. Отрицательные значения отсчитывают промежутки с конца строки. Для записи срезов используется следующий синтаксис:

последовательность[нач:кон:шаг]

где нач - промежуток начала среза, кон - конца среза, шаг - шаг. По умолчанию нач=0, кон=len(последовательность), шаг=1, если шаг не указан, второе двоеточие можно опустить.

Взятие элемента по индексу и срезы

А теперь пример работы со срезами:

```
>>> s = range(10)
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[0:3]
[0, 1, 2]
>>> s[-1:]
[9]
>>> s[::3]
[0, 3, 6, 9]
>>> s[0:0] = [-1, -1, -1]
>>> s
[-1, -1, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del s[:3]
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Как видно из этого примера, с помощью срезов удобно задавать любую подстроку, даже если она нулевой длины, как для удаления элементов, так и для вставки в строго определенное место.

Кортеж (tuple) - immutable

Для представления константных последовательностей используется тип кортеж. Для задания кортежей используются круглые скобки, но можно их и не указывать, если это не привносит неоднозначность.

```
point = (1.2, 3.4, 0.9)
point2 = 1, 3, 9
tuple1 = ('one', 1, [1])
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"
print("tup1[0]: {} ".format(tup1[0]))
print("tup2[1:5]: {}".format(tup2[1:5]))
```

Кортежи неизменяемы: нельзя изменить значение элемента кортежа, удалить часть кортежа, или наоборот, добавить новые элементы в кортеж. Однако, можно создавать новые кортежи из нескольких старых.

Функции для работы с кортежами

Ниже обобщены основные функции для работы с неизменяемыми последовательностями - кортежами.

Функция	Описание
len(s)	Длина последовательности s.
x in s	Проверка принадлежности элемента последовательности. В новых версиях Python можно проверять принадлежность подстроки строке. Возвращает True или False.
x not in s	= not x in s
s + s1	Конкатенация последовательностей
s*n или n*s	Последовательность из n раз повторенной s. Если n < 0, возвращается пустая последовательность.
s[i]	Возвращает i-й элемент s или len(s)-i-й, если i < 0
s[i:j:d]	Срез из последовательности s от i до j с шагом d. Так же как и для строк.
min(s)	Наименьший элемент s
max(s)	Наибольший элемент s

Примеры работы с кортежами

Примеры:

```
>>> t = (2, 2.05, "Hello")
>>> z, y, x = t
>>> print(z, y, x)
2 2.05 Hello
>>> x = 12,
>>> x
(12,)
>>> t = tuple('Hello!')
>>> t
('H', 'e', 'l', 'l', 'o', '!')
>>> a = 1
>>> b = 2
>>> a, b = b, a
>>> print(a, b)
2 1
>>> len((1,2,3))
3
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> ('Hi!',) * 4
('Hi!', 'Hi!', 'Hi!', 'Hi!')
>>> for x in (1, 2, 3): print x,
1 2 3
```

Множество (set) - mutable

Множество – это неупорядоченная коллекция уникальных хэшируемых объектов. Варианты использования: проверка на включение, гудаление дубликатов из последовательностей, выполнение математических операций (пересечение, объединение, разность множеств и строгая дизъюнкция).

Как и другие коллекции множества поддерживают операции: x in set, len(set), и for x in set.

Будучи неупорядоченной коллекцией, множество не содержит информации о позиции элемента или порядке его добавления. Соответственно, множество не поддерживает индексирование, срезы и другие операции свойственные последовательностям.

Т.к. множество – изменяемый тип, содержимое его может изменяться посредством методов add() и remove(). По той же причине для него не рассчитывается хэш, и оно не может быть ключом словаря или элементом другого множества

Однако, есть и неизменяемый аналог множества – тип frozenset (неизменяемое множество).

Операции над множествами

Функция	Описание
len(s)	Возвращает мощность множества *s*.
x in s	Проверяет *x* на включение в *s*.
x not in s	Проверяет, что *x* не входит в *s*.
.isdisjoint(*other*)	Возвращает True, если set не имеет общих элементов с *other*. Множества считаются непересекающимися, если и только если их пересечением является пустое множество.
issubset(*other*) set <= other	Проверяет, что каждый элемент set содержится также и в *other*.
set < other	Проверяет, что set – правильное подмножество *other*, т.е. set <= other и set != other.
issuperset(*other*) set >= other	Проверяет, что каждый элемент *other* содержится также и в set.
set > other	Проверяет, что set – правильное надмножество *other*, т.е., set >= other и set != other.
union(*other*, ...) set | other | …	Возвращает новое множество, состоящее из элементов set и всех others.
intersection(*other*, ...) set & other & ...	Возвращает новое множество, состоящее из общих элементов set и всех others.
difference(*other*, ...) set - other - ...	Возвращает новое множество, состоящее из элементов, которые есть только в set, но не в others.
symmetric_difference(*other*) set ^ other	Возвращает новое множество, состоящее из элементов, которые есть либо в set, либо в other, но не в обоих сразу.
copy()	Возвращает новое множество – поверхностную копию *s*.

Примеры работы с множествами

```
>>> a = set('hello')
>>> a
{'h', 'o', 'l', 'e'}

>>> a = {'a', 'b', 'c', 'd'}
>>> a
{'b', 'c', 'a', 'd'}

>>> a = {i ** 2 for i in range(10)}
>>> a
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> a = {} # а так нельзя, получается словарь
>>> type(a)
<class 'dict'>

>>> a = {1, 2, 3}
>>> b = {3, 4 ,5}
>>> a <= b
False
>>> a.intersection(b)
set([3])
```

Различие между set и frozenset

Единственное отличие set от frozenset заключается в том, что set - изменяемый тип данных, а frozenset - нет. Примерно похожая ситуация со списками и кортежами.

```
>>> a = set('qwerty')
>>> b = frozenset('qwerty')
>>> a == b
True
>>> type(a - b)
<class 'set'>
>>> type(a | b)
<class 'set'>
>>> a.add(1)
>>> b.add(1)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Словарь (dictionary) - mutable

Одним из сложных типов данных (наряду со строками и списками) в языке программирования Python являются словари. Словарь это изменяемый (как список) неупорядоченный (в отличие от строк и списков) набор пар "ключ:значение", где значение однозначно определяется по ключу.

Поскольку словарь является ассоциативным хэш-массивом, и хэш вычисляется для ключа при добавлении в словарь очередной пары - в качестве ключа должен использоваться неизменяемый (immutable) тип, чтоб не было необходимости в пересчете хэша при возможном изменении ключа.

Чтобы представление о словаре стало более понятным, можно провести аналогию с обычным словарем, например, англо-русским. На каждое английское слово в таком словаре есть русское слово-перевод: cat – кошка, dog – собака, table – стол и т.д. Если англо-русский словарь описывать с помощью Python, то английские слова будут ключами, а русские — их значениями:

{'cat':'кошка', 'dog':'собака', 'bird':'птица', 'mouse':'мышь'}

Способы задания словарей

Словарь можно задать разными способами. Что выведется на экран в результате выполнения следующего кода?

```
a = dict(one=1, two=2, three=3)
b = {'one': 1, 'two': 2, 'three': 3}
c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
d = dict([('two', 2), ('one', 1) , ('three', 3)])
e = dict({'three': 3, 'one': 1, 'two': 2})
print a == b == c == d == e
```

Еще способы:

```
>>> f = dict.fromkeys(['a', 'b'])
>>> f
{'a': None, 'b': None}
>>> g = dict.fromkeys(['a', 'b'], 2)
>>> g
{'a': 2, 'b': 2}
```

Функции для работы со словарями

Ниже обобщены основные функции для работы со словарями.

Функция	Описание
len(d)	Возвращает количество элементов в словаре d.
d[key]	Возвращает элемент словаря d с ключом key. Выбрасывает исключение KeyError, если ключа key в словаре нет.
d[key] = value	Записать в элемент d[key] значение value.
del d[key]	Удалить d[key] из d. Выбрасывает исключение KeyError, если ключа key в словаре нет.
key in d	Возвращает True если d содержит ключ key, иначе False.
key not in d	Эквивалентно not key in d.
iter(d)	Возвращает итератор по ключам словаря. Сокращенная форма для iterkeys()
fromkeys(seq[, value])	Создает словарь с ключами из последовательности seq и значениями value. fromkeys() – метод словаря, возвращающий новый словарь. Value, по умолчанию, равно None.
pop(key[, default])	Если ключ в словаре – удалить соответствующий элемент и вернуть его значение, иначе вернуть значение по умолчанию. Если значение по умолчанию не дано и ключ в словаре не найден, выбрасывается исключение KeyError.
setdefault(key[, default])	Если ключ в словаре – вернуть его значение, иначе добавить элемент с данным ключом и значением default и вернуть default. Default, по умолчанию, равно None.

Методы словарей

Метод	Описание
.clear()	Удаляет все элементы из словаря.
.copy()	Возвращает поверхностную копию словаря.
.get(key[, default])	Возвращает значение по ключу, если ключ в словаре, иначе default. Если default не задан, возвращается None, т.е. данный метод никогда не выбрасывает исключение KeyError
.has_key(key)	Проверка наличия ключа в словаре. has_key() устарел, вместо него используется key in d.
.items()	Возвращает копию списка элементов словаря – пар ключ:значение.
.iteritems()	Возвращает итератор по элементам словаря (парам ключ:значение).
.iterkeys()	Возвращает итератор по ключам словаря.
.itervalues()	Возвращает итератор по значениям словаря.
.keys()	Возвращает копию списка ключей словаря.
.popitem()	Удаляет и возвращает случайную пару (ключ:значение) из словаря.
.update([other])	Обновляет словарь парами ключ:значение из other, переписывая существующие ключи. Возвращает None.
.values()	Возвращает копию списка значений словаря.
.viewitems()	Возвращает view объект (ссылка только для чтения) для элементов словаря.
.viewkeys()	Возвращает view объект для ключей словаря.
.viewvalues()	Возвращает view объект для значений словаря.
.iter(dictview)	Возвращает итератор по view объектам для словаря.

Примеры работы со словарями

```
>>> d1 = {1: 1, 3: 8}
>>> d1[3]
8
>>> d1[2] = 4
>>> d1
{1: 1, 2: 4, 3: 8}
>>> d1[3] = 9
>>> d1
{1: 1, 2: 4, 3: 9}
>>> del d1[1]
>>> d1
{2: 4, 3: 9}
>>> 2 in d1
True
>>> 1 not in d1
True
```

Файл

Файл – это тип для работы с внешними данными, в самом простом случае – файлом на диске. Файловые объекты поддерживают базовые методы: read (), write (), readline (), readlines (), seek (), tell (), close ().

Файловые объекты имплементированы с использованием библиотеки stdio языка Си и могут быть созданы с помощью встроенной функции open(). Также файловые объекты могут быть возвращены другими встроенными функциями и методами, такими как os.popen(), os.fdopen() и makefile() методом socket объектов. Временные файлы могут быть созданы с использованием модуля tempfile, а высокоуровневые файловые операции, такие как копирование, перемещение и удаление файлов и директорий могут быть выполнены с помощью функций модуля shutil.

Когда файловая операция падает из-за проблем, связанных с вводом/выводом выбрасывается исключение IOError. Например, при попытке записи в файл открытый только для чтения.

Функции для работы с файлами

Функция	Описание
.open(*name*[, *mode*[, *buffering*]])	Открывает файл, возвращая объект типа Файл. Если файл нельзя открыть, выбрасывается исключение IOError. При открытии файла предпочтительнее использовать open(), чем вызывать конструктор файлового объекта напрямую. А еще лучше использовать менеджер контекста.
file.close()	Закрывает файл. Закрытый файл нельзя использовать для чтения или записи. Все подобные операции для закрытого файла приведут к выбрасыванию исключения ValueError. Вызов close() более одного раза для одного и того же фала допускается.
file.flush()	Очищает внутренний буфер, как операция fflush() у stdio. Не для всех файловых объектов.
file.fileno()	Возвращает целочисленный дескриптор файла, который используется низкоуровневой реализацией для запроса операции ввода/вывода у операционной системы. Может использоваться другими низкоуровневыми интерфейсами: fcntl module или os.read(),
file.isatty()	Возвращает True, если файл подключен к tty(-like) устройству, иначе False.
file.next()	Файловый объект является также итератором самого себя. Например, iter(f) возвращает f (хоть даже f закрыт). Когда файл используется как итератор, обычно в цикле for (например, for line in f: print line.strip()), метод next() вызывается на каждой итерации.
file.read([*size*])	Читает не более *size* байтов из файла (lменьше, если read встречает EOF до того как прочитала *size* байтов). Если аргумент *size* отрицательный или пропущен, читаются все данные до EOF. Байты возвращаются как строковый объект.

Функции для работы с файлами

Функция	Описание
file.readline([*size*])	Читает одну целую строку из файла. Завершающий символ перевода строки сохраняется в строке (но может быть пропущен, если файл заканчивается незавершенной строкой).
file.readlines([*sizehint*])	Читает до достижения EOF, используя readline(), и возвращает список прочитанных строк.
file.xreadlines()	Возвращает то же, что iter(f).
file.seek(*offset*[, *whence*])	Устанавливает текущую позицию в файле, как fseek() у stdio.
file.tell()	Возвращает текущую позицию в файле, как ftell() у stdio.
file.truncate([*size*])	Обрезает размер файла. Если аргумент *size* указан, файл обрезается по данный размер (не больше). По умолчанию *size* определяется текущей позицией.
file.write(*str*)	Пишет строку в файл. Ничего не возвращает. Из-за буферизации строка может не появиться в файле, пока не будут вызваны flush() или close().
file.writelines(*sequence*)	Пишет последовательность строк в файл. Последовательность моет быть итерируемым объектом, представляемым набором строк (обычно это список строк). Ничего не возвращает.

Работа с файлами

```
fo = open("myfile.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not: ", fo.closed
print "Opening mode: ", fo.mode
print "Softspace flag: ", fo.softspace
```

```
Name of the file: myfile.txt
Closed or not: False
Opening mode: wb
Softspace flag: 0
```

Атрибуты файлового объекта:

Атрибут	Описание
file.closed	Возвращает True если файл закрыт, иначе False.
file.mode	Возвращает режим доступа, с которым был открыт данный файл.
file.encoding	Кодировка, которую использует файл.
file.errors	Режим, в котором будут обрабатываться ошибки кодирования/декодирования.
file.name	Возвращает имя файла.
file.softspace	Возвращает False если при выводе содержимого файла следует отдельно добавлять пробел.
file.newlines	Если версия Python использует универсальный режим новых строк (по умолчанию) этот атрибут только-для-чтения существует и для файлов, открытых на чтение в этом же режиме, отслеживает типы новых строк, встреченные при чтении файла.

Режимы открытия файла

Modes	Description
'r'	открытие на чтение (является значением по умолчанию).
'w'	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый.
'x'	(с версии 3.3) открытие на запись, если файла не существует, иначе исключение.
'a'	открытие на дозапись, информация добавляется в конец файла.
'b'	открытие в двоичном режиме.
't'	(с версии 3.0) открытие в текстовом режиме (является значением по умолчанию).
'+'	открытие на чтение и запись

Режимы могут быть объединены, то есть, к примеру, 'rb' - чтение в двоичном режиме, ‘r+’ – чтение и запись в текстовом режиме.

Модули и импорты

Модули - это способ переиспользования кода, контейнер хранения. Модуль можно рассматривать как файл с кодом на Python. Чтоб наш код мог увидеть код из других модулей, надо эти модули импортировать с помощью ключевого слова import.

```
# импортируем модуль
# содержимое модуля доступно только через имя модуля
import math

print(math.pi)


# импортируем конкретный код из модуля (переменную, функцию)
# конкретный код доступен напрямую
from math import pi

print(pi)
```

Модули и импорты

```
# импортируем все содержимое модуля
# все содержимое модуля доступно напрямую
# так делать не следует!!!
# могут быть конфликты имен, теряется логическое разделение кода
from math import *

print(pi)


# импортируем конкретный код из модуля (переменную, функцию)
# и назначаем для него alias (псевдоним) для предотвращения
# возможных конфликтов имен и приведения импортированных
# имен к более удобному виду
from math import pi as new_pi

print(new_pi)
```

Области видимости

Область видимости некоторой именованной сущности языка Python (переменной, функции, класса) – это место в программном коде, где эта сущность была определена (т.е. ей было присвоено значение). При попытке использовать что-либо, что не попадает в текущую область видимости, выбрасывается исключение NameError.

В Python существует 4 области видимости:

- локальная (функция) - определение переменной (функции) выполняется внутри инструкции def;
- нелокальная (объемлющие функции) - определение переменной (функции) выполняется вне данной инструкции def, но внутри объемлющей инструкции def;
- глобальная (модуль) - определение переменной (функции) выполняется за пределами всех инструкций def в данном файле;
- встроенная область видимости (Python) – предопределенные имена в модуле встроенных имен.

Области видимости

```
x = 50  # глобальная переменная
def test_1(x):  # аргумент функции - локальная переменная
    print("Local arg: {}".format(x))
    x = 51  # переопределение локальной переменной
    print("Local var: {}".format(x))

test_1(x)  # передача глобальной переменной в качестве фактического параметра
print("Global var: {}".format(x)) # обращение к глобальной переменной


def test_2():
    # обращение к глобальной переменной: ошибки нет
    print("Global var: {}".format(x))

test_2()


def test_outer():
    x = 50  # для вложенной функции - это нелокальная переменная
    def test_inner():
        # обращение к нелокальной переменной: ошибки нет
        print("Nonlocal var: {}".format(x))
    test_inner()
    print("Local var: {}".format(x))

test_outer()
```

Области видимости

```
x = 50  # глобальная переменная

def test_2():
    # обращение к локальной переменной до ее определения
    # глобальная переменная здесь уже не видна: будет ошибка!!!
    print("Try global: {}".format(x))
    # определение локальной переменной, которое скрывает
    # глобальную переменную с таким же именем
    x = 51

test_2()


def test_outer():
    x = 50  # для вложенной функции - это нелокальная переменная
    def test_inner():
        # обращение к локальной переменной до ее определения
        # нелокальная переменная здесь уже не видна: будет ошибка!!!
        print("Try nonlocal: {}".format(x))
        # определение локальной переменной, которое скрывает
        # нелокальную переменную с таким же именем
        x = 51
    test_inner()
    print("Local var: {}".format(x))

test_outer()
```

global и nonlocal (для Python3)

При необходимости обратиться к глобальной или нелокальной переменной из функции, где эта переменная перекрыта локальной переменной, используются ключевые слова **global** и (для Python3) **nonlocal**.

```
x = 50  # глобальная переменная

def test_2():
    # явно указываем, что будем обращаться к глобальной переменной
    global x
    print("Global var: {}".format(x))
    x = 51  # переопределение глобальной переменной
test_2()


def test_outer():
    x = 50 # для вложенной функции - это нелокальная переменная
    def test_inner():
        # явно указываем, что будем обращаться к нелокальной переменной
        nonlocal x  # для Python2 надо указывать global
        print("Nonlocal var: {}".format(x))
        x = 51 # переопределение нелокальной переменной
    test_inner()
    print("Local var: {}".format(x))
test_outer()
```

Рекурсия

Рекурсия – это способность программы вызывать саму себя.

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)

x = 10
print("Factorial {}: {}".format(x, fact(x)))
```

Рекурсия

Чтоб понять, как будет выполняться рекурсивная функция, надо сысленно заменить вызовы этой функции на ее тело:

```
if 10 == 1:
    return 1
else:
    return 10 * (
        if 9 == 1:
            return 1
        else:
            return 10 * (
                if 8 == 1:
                    return 1
                else:
                    ...
                    return 10 * (
                        if 1 == 1:
                            return 1
                        else:
                            # эта ветка никогда не выполнится
```

Декораторы

Функции (как и все в Python) - это просто объекты, а значит их можно определять внутри других функций и передавать как аргумент.

```
def somefun(a, b):
    print('Got arguments {} and {}'.format(a, b))

myfun = somefun  # это НЕ вызов функции
myfun(20, 30)    # это вызов функции


def runfun(fun, *args):
    if len(args) == 2:
        fun(args[0], args[1]) # это вызов функции

# передача функции в качестве параметра в другую функцию
runfun(somefun, 50, 100)
```

Декораторы

Часто возникает необходимость дополнить код функции, не внося изменений в саму функцию. Это можно сделать путем оборачивания основной функции в другую функцию, реализующую необходимую функциональность. Примеры функций-оберток:

```
def benchmark(func):
    """
    Обертка для подсчета времени выполнения функции.
    """
    import time
    def wrapper(*args, **kwargs):
        t = time.clock()
        res = func(*args, **kwargs)
        print("{} spent {}".format(func.__name__, time.clock() - t))
        return res
    return wrapper

# оборачиваем somefun в benchmark
somefun = benchmark(somefun)
somefun(20, 30)
```

Декораторы

Пример функции-обертки:

```
def counter(func):
    """
    Обертка для подсчета количества вызовов функции.
    """
    def wrapper(*args, **kwargs):
        wrapper.count += 1
        res = func(*args, **kwargs)
        print("{} called {} times".format(func.__name__, wrapper.count))
        return res
    wrapper.count = 0
    return wrapper

# оборачиваем somefun в counter
somefun = counter(somefun)
somefun(20, 30)
```

Декораторы

Декоратор - это обертка над функцией в виде специальной конструкции (т.н. «синтаксический сахар»). Добавляется к определению функции. Предыдущие примеры можно записать с помощью декораторов.

```
@benchmark
def somefun(a, b):
    print('Got arguments {} and {}'.format(a, b))

@counter
def somefun(a, b):
    print('Got arguments {} and {}'.format(a, b))

@benchmark
@counter
def somefun(a, b):
print('Got arguments {} and {}'.format(a, b))

somefun(20, 30)

Got arguments 20 and 30
somefun called 1 times
wrapper spent 3.20740902933e-05
```

Практика

1. Напишите программу, которая выводит на экран числа от 1 до 100. При этом вместо чисел, кратных трем, программа должна выводить слово Fizz, а вместо чисел, кратных пяти — слово Buzz. Если число кратно пятнадцати, то программа должна выводить слово FizzBuzz.

2. Составить программу, которая будет считывать введённое пятизначное число. После чего, каждую цифру этого числа необходимо вывести в новой строке:
   Число: 10819
   1 цифра равна 1
   2 цифра равна 0
   3 цифра равна 8
   4 цифра равна 1
   5 цифра равна 9

3. Реализовать алгоритм сортировки выбором. Алгоритм состоит из следующих шагов:
   1. найти наименьший элемент в массиве
   2. поменять местами его и первый элемент в массиве
   3. найти следующий наименьший элемент в массиве
   4. и поменять местами его и второй элемент массива
   5. продолжать это пока весь массив не будет отсортирован

   arr = [0,3,24,2,3,7]
   // здесь реализованный алгоритм
   // на выходе должен получиться список, содержащий [0, 2, 3, 3, 7, 24]

Практика*

4. Реализовать функциональность, которая бы “сворачивала” и “разворачивала” символы табуляции в файле или строке. То есть, передается на вход файл или строка, необходимо заменить все символы табуляции на четыре пробела, либо же заменить все комбинации из четырех символов пробела на символ табуляции.

5. Интерполировать некие шаблоны в строке. Есть строка с определенного вида форматированием. необходимо заменить в этой строке все вхождения шаблонов на их значение из словаря.

6. * Есть список списков (матрица). Каждый внутренний список - это строка матрицы. Необходимо реализовать функцию, которая удаляет столбец, который содержит заданную цифру